

DWEC – Javascript Web Cliente.

JavaScript 00 – Lenguaje JavaScript

JavaScript 00 – Lenguaje JavaScript	1
Introducción	1
Usaremos JavaScript para:	2
Un poco de historia	2
Soporte en los navegadores	2
Herramientas	3
La consola del navegador	3
Editores	3
Editores on-line	3
npm	4
Git	4
GitHub	4
Incluir JavaScript en una página web	4
Mostrar información	6

Introducción

Para el desarrollo de las páginas web lo principal es un fichero HTML que contiene la información a mostrar en el navegador. Posteriormente surgió la posibilidad de “decorar” esa información para mejorar su apariencia, lo que dio lugar al CSS. Y también se pensó en dar dinamismo a las páginas, y apareció el lenguaje JavaScript. Que también lo escribiremos como Javascript o JS.

En un primer momento las 3 funcionalidades estaban mezcladas en el fichero HTML, pero eso complicaba bastante el poder leer esa página a la hora de mantenerla por lo que se pensó en separar los 3 elementos básicos:

HTML: se encarga de estructurar la página y proporciona su información, pero es una información estática

CSS: es lo que da forma a dicha información, permite mejorar su apariencia, permite que se adapte a distintos dispositivos, ...

JavaScript: es el que da vida a un sitio web y le permite reaccionar a las acciones del usuario

Por tanto, nuestras aplicaciones tendrán estos 3 elementos y lo recomendable es que estén separados en distintos ficheros:

El HTML lo tendremos habitualmente en un fichero index.html, normalmente en una carpeta llamada public, html (o similar).

El CSS lo tendremos en uno o más ficheros con extensión .css dentro de una carpeta llamada styles, estilos, css (o similar).

El JS estará en ficheros con extensión .js en un directorio llamado js, scripts, funciones (o similar).

Nota: Existen variedades y complementos del lenguaje JavaScript, como son TypeScript, CoffeScript, Vue, Angular, etc.

Vanilla JavaScript es como se conoce al lenguaje JavaScript cuando se utiliza sin ninguna librería o *framework*. La traducción más castellana sería “JavaScript a pelo”.

Las características principales de Javascript son:

- Es un lenguaje interpretado, no compilado.
- Se ejecuta en el lado cliente (en un navegador web), aunque hay implementaciones como NodeJS para el lado servidor.
- Es un lenguaje orientado a objetos (podemos crear e instanciar objetos y usar objetos predefinidos del lenguaje) pero basado en prototipos (por debajo, un objeto es un prototipo y nosotros podemos crear objetos sin instanciarlos, haciendo copias del prototipo).
- Se trata de un lenguaje débilmente tipado, con tipificación dinámica (no se indica el tipo de datos de una variable al declararla e incluso puede cambiarse).

Usaremos JavaScript para:

- Cambiar el contenido de la página
- Cambiar los atributos de un elemento
- Cambiar la apariencia de algo
- Validar datos de formularios
- ...

Sin embargo, por razones de seguridad, JavaScript no nos permite hacer cosas como:

- Acceder al sistema de ficheros del cliente
- Capturar datos de un servidor (puede pedirlo y el servidor se los servirá, o no)
- Modificar las preferencias del navegador
- Enviar e-mails de forma invisible o crear ventanas sin que el usuario lo vea
- ...

Un poco de historia

JavaScript es una implementación del lenguaje **ECMAScript** (el estándar que define sus características).

El lenguaje surgió en 1997 y todos los navegadores a partir de 2012 soportan al menos la versión **ES5.1** completamente.

En 2015 se lanzó la 6^a versión, inicialmente llamada **ES6** y posteriormente renombrada como **ES2015**, que introdujo importantes mejoras en el lenguaje y que es la versión que usaremos nosotros.

Desde entonces van saliendo nuevas versiones cada año que introducen cambios pequeños. La última es la versión 13, llamada ES2022, que ha sido aprobada en verano de 2022.

Las principales mejoras que introdujo ES2015 son: clases de objetos, let, for..of, Map, Set, Arrow functions, Promesas, spread, destructuring, ...

Soporte en los navegadores

Los navegadores no se adaptan inmediatamente a las nuevas versiones de JavaScript, por lo que puede ser un problema usar una versión muy moderna e JS ya que puede haber partes de los programas que no funcionen en los navegadores de muchos usuarios.

En la página de [Kangax](#) podemos ver la compatibilidad de los diferentes navegadores con las distintas versiones de JavaScript.

También podemos usar [CanIUse](#) para buscar la compatibilidad de un elemento concreto de JavaScript, así como de HTML5 o CSS3.

Si queremos asegurar la máxima compatibilidad debemos usar la versión ES5 (pero nos perdemos muchas mejoras del lenguaje).

Lo mejor sería usar la ES6 (o posterior) y después *transpilar* nuestro código a la versión ES5. De esto se ocupan los *transpiladores* (**Babel** es el más conocido), por lo que no suponen un esfuerzo extra para el programador.

Si queremos asegurar la máxima compatibilidad debemos usar la versión ES5 (pero nos perdemos muchas mejoras del lenguaje) o mejor, usar la ES6 (o posterior) y después transpilar nuestro código a la versión ES5. De esto se ocupan los transpiladores (Babel es el más conocido) por lo que no suponen un esfuerzo extra para el programador.

Herramientas

La consola del navegador

Es la herramienta que más nos va a ayudar a la hora de depurar nuestro código. Abrimos las herramientas para el desarrollador (en Chrome y Firefox pulsando la tecla F12) y vamos a la pestaña Consola:

Allí vemos mensajes del navegador como errores y advertencias que genera el código y, todos los mensajes que pongamos en el código para ayudarnos a depurarlo (usando los comandos `console.log` y `console.error`).

Además, en ella podemos escribir instrucciones JavaScript que se ejecutarán mostrando su resultado. También la usaremos para mostrar el valor de nuestras variables y para probar código que, una vez que funcione correctamente, lo copiaremos a nuestro programa.

Siempre depuraremos los programas desde la consola (pondremos puntos de interrupción, veremos el valor de las variables, ...).

Editores

Podemos usar el que más nos guste, desde editores tan simples como NotePad++ hasta complejos IDEs. La mayoría soportan las últimas versiones de la sintaxis de JavaScript (Netbeans, Eclipse, Visual Studio, Sublime Text, Atom, Kate, Notepad++, ...).

Por el momento utilizaremos el editor **Visual Studio Code** por su sencillez y por los *plugins* que incorpora para hacer más cómodo el trabajo del desarrollador. En *Visual Studio Code* instalaremos algunos *plugins* como:

SonarLint: es más que un *linter* y nos informa de todo tipo de errores, pero también del código que no cumple las recomendaciones (incluye gran número de reglas). Marca el código mientras lo escribimos y además podemos ver todas las advertencias en el panel de Problemas (Ctrl+Shift+M)

Otros plugins: según el documento a nuestra disposición “Instalación de Extensiones de Visual Studio Code”

Editores on-line

Son muy útiles porque permiten ver el código y el resultado a la vez. Normalmente tienen varias pestañas o secciones de la página donde poner el código HTML, CSS y Javascript y ver su resultado.

Algunos de los más conocidos son [Codesandbox](#), [Fiddle](#), [Plunker](#), [CodePen](#), ...aunque hay muchos más.

Como ejemplo: <https://jsfiddle.net/tqobL253/>

Que se puede ver cómo queda incrustado en una página web si incrustamos el siguiente código:

```
<script async src="//jsfiddle.net/tqobL253/embed/js,html,css,result/dark/"></script>
```

npm

npm es el gestor de paquetes del framework JavaScript **Node.js** y suele utilizarse en programación *frontend* como gestor de dependencias de la aplicación.

Esto significa que será la herramienta que se encargará de descargar y poner a disposición de nuestra aplicación todas las librerías JavaScript que vayamos a utilizar.

Para instalar *npm* tenemos que instalar *NodeJS*.

Git

Usaremos repositorios git realizar el control de versiones de nuestras aplicaciones.

GitHub

Usaremos una cuenta en GitHub.com, generalmente asociada a otra cuenta de Gmail.com, para llevar el control de las versiones y tener copia de nuestro código en la nube.

Incluir JavaScript en una página web

El código JavaScript va entre etiquetas `<script>`. Puede ponerse en el `<head>` o en el `<body>`. Funciona como cualquier otra etiqueta.

El navegador la interpreta cuando llega a ella. Va leyendo la etiqueta y ejecutando el fichero línea a línea.

Opción 1: en HEAD

```
<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <script>
    function saludar() {
      alert("Hola majos/as");
    }
    saludar();
  </script>
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>
</body>
</html>
```

Opción 2: en BODY (delante de otros elementos):

```
<!DOCTYPE html>
<head>
```

```

<meta charset="UTF-8">
</head>
<body>
  <script>
    function saludar() {
      alert("Hola majos/as");
    }
    saludar();
  </script>
  <h1>Bienvenidos a DWEC</h1>
</body>
</html>

```

Opción 3: en fichero externo de extensión js (en HEAD)

```

<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <script src="js/funciones.js"></script>
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>

  <h1>hasta otra</h1>
</body>
</html>

```

Opción 4 (Recomendada): en fichero externo de extensión js (en BODY)

```

<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
  <h1>Bienvenidos a DWEC</h1>

  <h1>hasta otra</h1>
  <script src="js/funciones.js"></script>
</body>
</html>

```

Lo mejor, en cuanto a rendimiento, es ponerla al final del `<body>` para que no se detenga el renderizado de la página mientras se descarga y se ejecuta el código.

También podemos ponerlo en el `<head>` pero usando los atributos **async** y/o **defer** (en Internet encontraréis mucha información sobre esta cuestión, por ejemplo [aquí](#)).

Es posible poner el código directamente entre la etiqueta `<script>` y su etiqueta de finalización, pero lo correcto es que esté en un fichero externo (con extensión **.js**) que cargamos mediante el atributo `src` de la etiqueta. Así conseguimos que la página HTML cargue más rápido que si lo ponemos al final del BODY o usamos `async`.

Además, es preferible no mezclar HTML y JS en el mismo fichero, lo que mejora la legibilidad del código y facilita su mantenimiento.

```
<script src="./scripts/main.js"></script>
```

Mostrar información

JavaScript permite mostrar al usuario ventanas modales para pedirle o mostrarle información. Las funciones que lo hacen son:

- `window.alert(mensaje)`: Muestra en una ventana modal *mensaje* con un botón de *Aceptar* para cerrar la ventana.
- `window.confirm(mensaje)`: Muestra en una ventana modal *mensaje* con botones de *Aceptar* y *Cancelar*. La función devuelve **true** o **false** en función del botón pulsado por el usuario.
- `window.prompt(mensaje [, valor predeterminado])`: Muestra en una ventana modal *mensaje* y debajo tiene un campo donde el usuario puede escribir, junto con botones de *Aceptar* y *Cancelar*. La función devuelve el valor introducido por el usuario como texto (es decir que si introduce 54 lo que se obtiene es "54") o **false** si el usuario pulsa *Cancelar*.

También se pueden escribir las funciones sin *window*. (es decir `alert('Hola')` en vez de `window.alert('Hola')`) ya que en JavaScript todos los métodos y propiedades de los que no se indica de qué objeto son se ejecutan en el objeto *window*.

Si queremos mostrar una información para depurar nuestro código no utilizaremos `alert(mensaje)` si no `console.log(mensaje)` o `console.error(mensaje)`. Estas funciones muestran la información en la consola del navegador. La diferencia es que `console.error` la muestra como si fuera un error de JavaScript.

Por último, indicar que podremos usar el método `write` del objeto `document` para enviar información (en lenguaje HTML) al documento HTML donde ponemos este código. `Document.window(código_html)`. Ejemplos: `document.write("Hola Daw2")` o `document.write('<p>Hola Daw2</p>')`



DWEC - Javascript Web Cliente.

JavaScript 01 A – Sintaxis (I)

JavaScript 01 – Sintaxis	1
Variables	1
Use Strict.....	2
Otro ejemplo de ámbito de variables:.....	3
Variables locales y variables globales.....	4
Variables constantes (const)	5
Funciones.....	5
Parámetros.....	5
Funciones anónimas.....	7
Arrow functions (funciones flecha)	7
Estructuras y bucles.....	8
Estructura condicional: if	8
Estructura condicional: switch	9
Bucle while	9

Variables

Javascript es un lenguaje débilmente tipado. Esto significa que no se indica de qué tipo es una variable al declararla, incluso puede cambiar su tipo a lo largo de la ejecución del programa. Ejemplo:

```
let miVariable;      // declaro miVariable y como no se asignó un valor valdrá undefined
miVariable='Hola';  // ahora su valor es 'Hola', por tanto contiene una cadena de texto
miVariable=34;       // pero ahora contiene un número
miVariable=[3, 45, 2]; // y ahora un array
miVariable=undefined; // para volver a valer el valor especial undefined
```

EJERCICIO: Ejecuta en la consola del navegador las instrucciones anteriores y comprueba el valor de **miVariable** tras cada instrucción (para ver el valor de una variable simplemente ponemos en la consola su nombre: `miVariable`

Ni siquiera estamos obligados a declarar una variable antes de usarla, aunque es recomendable para evitar errores que nos costará depurar.

Las variables se declaran con **let** (lo recomendado desde ES2015), aunque también pueden declararse con **var**.

La diferencia es que con **let** la variable sólo existe en el bloque en que se declara.

Mientras que con **var** el **ámbito (scope)** de la variable es global, se extiende. Es decir, existe en toda la función o ámbito en el que se declara:

```

if (edad > 18) {
    let textoLet = 'Eres mayor de edad';
    var textoVar = 'Eres mayor de edad';
} else {
    let textoLet = 'Eres menor de edad';
    var textoVar = 'Eres menor de edad';
}
console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar); // mostrará la cadena

```

Cualquier variable que no se declara dentro de una función (o si se usa sin declarar) es *global*. Debemos siempre intentar NO usar variables globales.

Además, como podemos comprobar con el siguiente ejemplo, las variables declaradas con var pueden duplicarse sin que se produzca error, y que a la larga puede acarrear muchos problemas.

```

// CREACIÓN DE VARIABLES CON LET

/* let declara una variable limitando su ámbito (scope) al bloque,
   declaración o expresión donde se está usando. */

// SINTAXIS: let nombreVariable [= valor];

var persona = "Santi";
var persona = "Profesor";
console.log (persona);

let persona2 = "Ana";
//let persona2 = "Jefa de Estudios"; //Esta instrucción devuelve un error
console.log (persona2);

```

Se recomienda que los nombres de las variables sigan la sintaxis *camelCase* (ej.: *miPrimeraVariable*).

Desde ES2015 también podemos declarar constantes con **const**. Se les debe dar un valor al declararlas y si intentamos modificarlo posteriormente se produce un error.

NOTA: en la página de [Babel](#) podemos teclear código en ES2015 y ver cómo quedaría una vez *transpilado* a ES5.

Cuando veamos “next generation JavaScript” se refiere a la próxima versión de Javascript pendiente de ser aceptada

Use Strict

Para evitar problemas futuros, y que no se produzca algún error si no declaramos una variable, incluiremos al principio de nuestro código la instrucción “use estrict”, que nos obliga a declarar las variables.

```
'use strict';
```

Veamos un ejemplo:

```

// USE STRICT O MODO ESTRICTO

/* "use strict" es una línea que indica que el código debe ser usado "en modo estricto",
   es decir, no se pueden utilizar variables no declaradas.
   Fuera de una función tiene ámbito global; dentro de ella, local (el de la función).
*/

```

```
// SINTAXIS: "use strict";

// "use strict"; // si aplicamos este código da error al no declarar persona2

persona2 = "Santi";
let nacimiento;

function informar (){
    "use strict";
    let persona = "Santi";
    nacimiento = "1915";
    console.log(persona + " nacio en "+nacimiento);
}

informar();
```

Otro ejemplo de ámbito de variables:

Caso 1º: Usamos dos variables (a) definidas con let, cada una queda circunscrita al ámbito donde ha sido creada.

/ ÁMBITO DE VARIABLES

```
/* El ámbito de una variable (scope) es la zona del programa en la que se define.
Javascript define dos ámbitos para variables: local y global.
Mediante var podemos definir como ámbito local el ámbito de una función.
Con let, por el contrario, podemos diferenciar también el ámbito de bloque. */
```

```
function verAmbito(){
    "use strict";
    let a = "Ámbito de función";
    if (true){
        let a = "Ámbito de bloque";
        console.log ("El ámbito de bloque a es: "+a);
    }
    console.log ("El ámbito de función a es: "+a);
}
verAmbito();
```

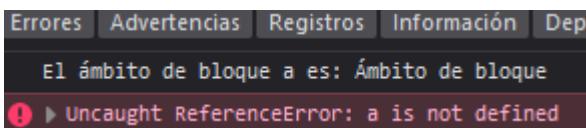
Errores | Advertencias | Registros | Información | Depura
 El ámbito de bloque a es: Ámbito de bloque
 El ámbito de función a es: Ámbito de función
 »

Si en el caso 1º cambiamos el **let** (dentro del if) por **var**, da error de variable ya declarada, puesto que ahora var extiende su scope a toda la función.

top ▾ | Filtrar
 ✘ Uncaught SyntaxError: Identifier 'a' has already been declared

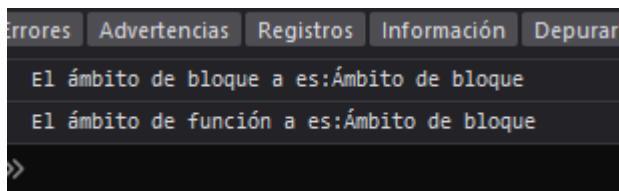
Caso 2º: Usamos una variable (a) definida con **let** dentro de un **if**, su ámbito queda reducido al bloque if, por tanto, fuera del bloque daría error.

```
function verAmbito(){
  "use strict";
  if (true){
    let a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es: "+a);
  }
  console.log ("El ámbito de función a es: "+a); // produce error
}
verAmbito();
```



Caso 3º: Usamos una variable (a) definida con **var** dentro de un **if**, su ámbito se extiende a toda la función.

```
function verAmbito(){
  "use strict";
  //let a = "Ámbito de función"; // SIMILAR AL CASO: var a = "Ámbito de función";
  if (true){
    var a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es:"+a);
  }
  console.log ("El ámbito de función a es:"+a);
}
verAmbito();
// console.log ("El ámbito de función a es: "+a); // ERROR, pues no llega el ámbito
```



Nota: si en el caso 3º cambiamos var por let (estamos igual que el caso)

Conclusiones: Se pueden trabajar con **var** y **let** a la vez, pero sería peligroso. En la medida de lo posible trabajaremos con **let**, e incluso con **const** cuando sepamos que esa variable nunca debe cambiar. Además, debemos pensar en que al usar **const** y **let** (su ámbito queda reducido a nuestros bloques) nunca interferirán con otras funciones o código de otro programador en nuestra aplicación.

A modo de repaso recordemos que en este lenguaje de programación no es obligado declarar las variables.

En el caso de ECMAScript 5 usamos la palabra reservada **var**.

En el caso de ECMAScript 6+ además de **var** también podemos usar **let** y **const**.

Variables locales y variables globales

Una variable global se puede usar en cualquier lugar del script.

Una variable local sólo tiene validez dentro del ámbito de una función.

Las normas que aplican para definir el tipo de variable que estamos usando se resumen con 3 normas sencillas:

- Cualquier variable declarada o no declarada en la raíz de un script es siempre de ámbito global.
- Cualquier variable declarada dentro de una función es de ámbito local.
- Una variable NO declarada dentro de una función adquiere ámbito global.

En caso de duda, que es cuando la variable no ha sido declarada en un ámbito determinado, la variable adquiere comportamiento global.

Casos de uso de las propiedades del ámbito de las variables

En javascript se pueden dar algunos casos curiosos:

- Dentro de una función podemos sobrescribir el valor de una variable global. Esto puede suceder por ejemplo por error si nos hemos olvidado de declarar una variable... y cargarnos un dato externo a la función sin darnos cuenta.
- Cuando declaramos una variable local con el mismo nombre que una variable global, temporalmente la variable local tiene prioridad sobre la global. Dentro de la función usaremos la variable local. Deberemos tratarlas como si temporalmente la variable global hubiera dejado de existir para nosotros.
- Hay que recordar que una variable declarada dentro de los paréntesis de declaración de la función se considera declarada de tipo local y que por cuestiones de sintaxis nunca va acompañada de la palabra reservada `let`.

Variables constantes (`const`)

Las variables constantes en Javascript (`const`) tienen ámbito de bloque al igual que las variables definidas utilizando `let`. Es importante tener en cuenta que el valor de una constante no puede variar (reasignarse), por tanto, se asignan en el momento en que se declaran. Para diferenciarlo de las variables conviene utilizar **TODOMAYÚSCULAS**.

Funciones

Se declaran con la palabra reservada **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene `return` es como si devolviera `undefined`).

Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado *hoisting*: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

EJERCICIO mensaje: Realiza una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

Parámetros

Si se llama a una función con menos parámetros de los declarados, el valor de los parámetros no pasados será `undefined`:

```
function potencia(base, exponente) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra undefined
    let valor=1;
```

```

    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

let resultado = potencia(4); // devolverá 1 ya que no se ejecuta el for
console.log(resultado);
//console.log(potencia(4)); // devolverá 1 ya que no se ejecuta el for

```

Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```

function potencia(base, exponente=2) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra 2 la primera vez y 5 la segunda
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

console.log(potencia(4));       // mostrará 16 (4^2)
console.log(potencia(4,5));     // mostrará 1024 (4^5)

```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía:

```

function potencia(base, exponente) {
    exponente = exponente || 2; // si exponente vale undefined se la asigna el valor 2
    ...
}

```

También es posible acceder a los parámetros desde el array **arguments[]** por si no sabemos cuántos parámetros recibiremos:

```

function sumar () {
    let result = 0;
    for (let i=0; i<arguments.length; i++)
        result += arguments[i];
    return result;
}

console.log(sumar(4, 2));           // mostrará 6
console.log(sumar(4, 2, 5, 3, 2, 1, 3)); // mostrará 20

```

En Javascript las funciones son un tipo de datos más, por lo que podemos hacer cosas como pasárlas como argumento o asignárlas a una variable:

```

const cuadrado=function(value){
    return value * value;
}
function aplicarFuncion(dato, funcion_a_aplicar){

```

```

        return funcion_a_aplicar(dato);
    }
let resultado = aplicarFuncion(3,cuadrado); // devolverá 9 (3^2) y lo guarda en resultado

```

A este tipo de funciones (en nuestro caso la función *cuadrado*) se llaman *funciones de primera clase* y son típicas de lenguajes funcionales.

Un lenguaje de programación se dice que tiene **Funciones de primera clase** cuando las funciones en ese lenguaje son tratadas como cualquier otra variable. Por ejemplo, en ese lenguaje, una función puede ser pasada como argumento a otras funciones, puede ser retornada por otra función y puede ser asignada a una variable.

Funciones anónimas

Como acabamos de ver, podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignarse a un manejador de eventos. Ejemplo:

```

let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();           // se ejecuta la función

```

Como vemos, asignamos una función a una variable de forma que podamos “ejecutar” dicha variable.

Arrow functions (funciones flecha)

ES2015 permite declarar una función anónima de forma más corta. Ejemplo sin *arrow function*:

```

let potencia = function(base, exponente) {
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

```

Al escriirla con la sintaxis de una *arrow function* lo que hacemos es:

- Eliminamos la palabra *function*
- Si sólo tiene 1 parámetro podemos eliminar los paréntesis de los parámetros
- Ponemos el símbolo =>
- Si la función sólo tiene 1 línea podemos eliminar las {} y la palabra *return*

El ejemplo con *arrow function*:

```

let potencia = (base,exponente) => {
    let valor=1;
    for (let i=1; i<= exponente; i++){
        valor= valor * base
    }
    return valor
}

```

Otra forma de la función anterior con *arrow function*:

```
let resultado = (base, exponente) => base ** exponente;
```

Otro ejemplo, sin *arrow function*:

```
let cuadrado= function(base) {
    return base * base;
}
```

con *arrow function*:

```
let cuadrado = (base) => base * base; //puedes eliminar los paréntesis porque hay un único argumento
```

EJERCICIO: Haz una *arrow function* que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la “traduces” a *arrow function*.

Estructuras y bucles

Estructura condicional: if

El **if** es como en la mayoría de los lenguajes. Puede tener asociado un **else** y pueden anidarse varios con **else if**.

```
if (condicion) {
    ...
} else if (condicion2) {
    ...
} else if (condicion3) {
    ...
} else {
    ...
}
```

Ejemplo:

```
if (edad < 18) {
    console.log('Es menor de edad');
} else if (edad > 65) {
    console.log('Está jubilado');
} else {
    console.log('Edad correcta');
}
```

Operador ternario

Se puede usar el operador ternario **? :** que es como un *if* que devuelve un valor:

```
let edad=14;
let esMayorDeEdad = edad > 18 ? true : false;
console.log(esMayorDeEdad);
```

Estructura condicional: switch

El **switch** también es como en la mayoría de los lenguajes. Hay que poner *break* al final de cada bloque para que no continúe evaluando:

```
switch(color) {
    case 'blanco':
    case 'amarillo': // Ambos colores entran aquí
        colorFondo='azul';
        break;
    case 'azul':
        colorFondo='amarillo';
        break;
    default:          // Para cualquier otro valor
        colorFondo='negro';
}
```

Javascript permite que el *switch* en vez de evaluar valores pueda evaluar expresiones. En este caso se pone como condición *true*:

```
switch(true) {
    case age < 18:
        console.log('Eres muy joven para entrar');
        break;
    case age < 65:
        console.log('Puedes entrar');
        break;
    default:
        console.log('Eres muy mayor para entrar');
}
```

Bucle while

Podemos usar el bucle *while...do*

```
while (condicion) {
    // sentencias
}
```

que se ejecutará 0 o más veces. Ejemplo:

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota) {
    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar");
}
```

Si queremos asignar más de una **condición** al bucle, usaremos los operadores lógicos. En el siguiente ejemplo, el bucle finaliza cuando se pulsa cancelar o se introduce el valor **cero**.

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota && nota!=0) {
```

```

    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar)");
}
console.log("última nota: "+ nota);

```

O el bucle do...while:

```

do {
    // sentencias
} while (condicion)

```

que al menos se ejecutará 1 vez. Ejemplo:

```

let nota;
do {
    nota=prompt('Introduce una nota (o cancela para finalizar)');
    console.log('La nota introducida es: '+nota);
} while (nota)

```

EJERCICIO adivina: Haz un programa para que el usuario juegue a adivinar un número. Obtén un número al azar (busca por internet cómo se hace o simplemente guarda el número que quieras en una variable) y ve pidiendo al usuario que introduzca un número. Si es el que busca, le dices que lo ha encontrado y si no le mostrarás si el número que busca es mayor o menor que el introducido. El juego acaba cuando el usuario encuentra el número o cuando pulsa en ‘Cancelar’ (en ese caso le mostraremos un mensaje de que ha cancelado el juego).



DWEC - Javascript Web Cliente.

JavaScript 01 B – Sintaxis (II)

JavaScript 01 B – Sintaxis (II).....	1
Bucle: for.....	1
Bucle: for con contador.....	1
Bucle: for...in	1
Bucle: for...of	2

Bucle: for

Tenemos muchos *for* que podemos usar.

Bucle: for con contador

Creamos una variable **contador** que controla las veces que se ejecuta el *for*:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let i=0; i<datos.length; i++) {
    sumaDatos += datos[i];
}
// El valor de sumaDatos será 125
```

EJERCICIO: El factorial de un número entero n es una operación matemática que consiste en multiplicar ese número por todos los enteros menores que él: **n x (n-1) x (n-2) x ... x 1**. Así, el factorial de 5 (se escribe 5!) vale **5! = 5 x 4 x 3 x 2 x 1 = 120**. Haz un script que calcule el factorial de un número entero positivo.

Bucle: for...in

El bucle se ejecuta una vez para cada elemento del array (o propiedad del objeto) y se crea una variable **índice** que toma como valores la posición del elemento en el array:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let indice in datos) {
    sumaDatos += datos[indice];      // Los valores que toma indice son 0, 1, 2, 3
}
// El valor de sumaDatos será 125
```

También sirve para recorrer las propiedades de un objeto:

```

let profe={
    nom:'Santi',
    ape1:'Blanco',
    ape2:'Arenal'
}

let nombreCompleto='';

for (let campo in profe) {
    nombreCompleto += profe[campo] + ' ';
}
console.log(nombreCompleto);
// El valor de nombreCompleto será 'Santiago Blanco Arenal'

```

Bucle: for...of

Es similar al `for...in` pero la variable contador en vez de tomar como valor cada índice toma cada elemento. Es nuevo en ES2015:

```

let sumaDatos = 0;

for (let valor of datos) {
    sumaDatos += valor;      // los valores que toma valor son 5, 23, 12, 85
}
// El valor de sumaDatos será 125

```

También sirve para recorrer los caracteres de una cadena de texto:

```

let cadena = 'Hola';

for (let letra of cadena) {
    console.log(letra);      // los valores de letra son 'H', 'o', 'l', 'a'
}

```

EJERCICIO: Haz 3 funciones a las que se le pasa como parámetro un array de notas y devuelve la nota media. Cada una usará un `for` de una de las 3 formas vistas. Pruébalas en la consola.



DWEC - Javascript Web Cliente.

JavaScript 01 – Sintaxis (III)

JavaScript 01 – Sintaxis (III)	1
Tipos de datos básicos	1
Casting de variables	2
Number	2
Veamos algunos ejemplos de código y la salida en consola:.....	3
Otras funciones útiles son:.....	4
String.....	4
Template literals	5
Tratamiento de una cadena como un array	5
Boolean	6

Tipos de datos básicos

Para saber de qué tipo es el valor de una variable tenemos el operador **typeof**. Ej.:

- `typeof 3` devuelve *number*
- `typeof 'Hola'` devuelve *string*

En Javascript hay 2 valores especiales:

- **undefined**: es lo que vale una variable a la que no se ha asignado ningún valor.
- **null**: es un tipo de valor especial que podemos asignar a una variable. Es como un objeto vacío (`typeof null` devuelve *object*)

También hay otros valores especiales relacionados con operaciones numéricas (o con números):

- **NaN (Not a Number)**: indica que el resultado de la operación no puede ser convertido a un número (ej. `'Hola'*2`, aunque `'2'*2` daría 4 ya que se convierte la cadena '2' al número 2)
- **Infinity y -Infinity**: indica que el resultado es demasiado grande o demasiado pequeño (ej. `1/0` o `-1/0`)

```
js > JS funciones.js > ...
You, hace 33 segundos | 2 authors (You and others)
1 'use strict'; Profesor, anteal
2 console.log("tareaxxx");
3 console.log(typeof(3));
4 console.log(typeof("pepe"));
5 let a;
6 console.log(typeof(a));
7 console.log(typeof(null));
8 console.log("hola"*2);
9 console.log("3"*2);
10 console.log(-2/0);
11 console.log(typeof(-2/0));
12
```

Group similar messages in console Show CORS errors in console Treat code evaluation as user action

tareaxxx	funciones.js:2
number	funciones.js:3
string	funciones.js:4
undefined	funciones.js:6
object	funciones.js:7
NaN	funciones.js:8
6	funciones.js:9
-Infinity	funciones.js:10
number	funciones.js:11

Casting de variables

Como hemos dicho, las variables pueden contener cualquier tipo de valor.

En las operaciones, Javascript realiza **automáticamente** las conversiones necesarias para, si es posible, realizar la operación.

Ejemplos:

- '4' / 2 → devuelve 2 (convierte '4' en 4 y realiza la operación)
- '23' - null → devuelve 23 (hace 23 - 0)
- '23' - undefined → devuelve NaN (no puede convertir undefined a nada, así que no puede hacer la operación)
- '23' * true → devuelve 23 (23 * 1)
- '23' * 'Hello' → devuelve NaN (no puede convertir 'Hello')
- 23 + 'Hello' → devuelve '23Hello' (+ es el operador de concatenación, así que convierte 23 a '23' y los concatena)
- 23 + '23' → devuelve 2323 (OJO, convierte 23 a '23', no al revés)

Ten en cuenta que en Javascript todo son objetos, por lo que todo tiene métodos y propiedades. Veamos brevemente los tipos de datos básicos.

EJERCICIO: Prueba en la consola las operaciones anteriores y alguna más con la que tengas dudas de qué devolverá.

Number

Sólo hay 1 tipo de números, no existen enteros y decimales. El tipo de dato para cualquier número es **number**. El carácter para la coma decimal es el . (como en inglés, así que 23,12 debemos escribirlo como 23.12).

Tenemos los operadores aritméticos +, -, *, /, ** y % y los unarios ++ y --

Existen los valores especiales **Infinity** y **-Infinity** (23/0 no produce un error sino que devuelve *Infinity*).

Podemos usar los operadores aritméticos junto al operador de asignación = (=, -=, *=, /= y %=).

Cuidado!! Observa la diferencia en los ejemplos de abajo.

The screenshot shows two examples in a browser's developer tools console. In the first example, line 14 contains a syntax error: 'x+=y;' instead of 'x=y+'. The output shows 'NaN' because the variable 'x' was never assigned a value before the addition attempt. In the second example, line 13 has 'let x=0;' with a red box around it, and line 14 has 'x+=y;'. The output shows '5' with a red box around it, demonstrating that the variable 'x' was correctly initialized to 0 before the addition operation.

```

12 let y=5;
13 let x;
14 x+=y;      You, hace 3 segundos .
15 console.log(x);                                NaN
                                                               funciones.js:15

12 let y=5;
13 let x=0;
14 x+=y;
15 console.log(x);                                5
                                                               funciones.js:15
  
```

Algunos métodos útiles de los números son:

- **.toFixed(num)**: redondea el número a los decimales indicados. Ej. 23.2376.toFixed(2) devuelve 23.24
- **.toLocaleString()**: devuelve el número convertido al formato local. Ej. 23.76.toLocaleString() devuelve '23,76' (Los navegadores en español convierten el punto decimal en coma y ponen el punto separador de miles)

16	<code>let x=12536.231265;</code>	12536.23	funciones.js:17
17	<code>console.log (x.toFixed(2));</code>		
18	<code>console.log (x.toLocaleString());</code>	12.536,231	funciones.js:18

Podemos forzar la conversión a número con la función **Number(valor)**. Ejemplo `Number('23.12')` devuelve 23.12

Veamos algunos ejemplos de código y la salida en consola:

```
let num=23.33333333333;
console.log(typeof(num));
console.log(num);
```

```
num= 45212.44444
console.log(typeof(num));
console.log(num.toLocaleString());
```

```
let num2 = 36.2222;
console.log(typeof(num2));
num2=num2.toLocaleString();
console.log(typeof(num2));
console.log(num2);
```

```
num2=888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
```

```
num2=888.8888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
```

En el caso anterior, no es numérico porque después de convertir `toLocaleString()`, no puede volver a numérico.

Si se convierte con el método `toString()`, sí que puede retornar a numérico, como vemos en el siguiente ejemplo:

The screenshot shows a browser's developer tools console. On the left, there is a block of JavaScript code:

```

num2=888.8888; //ojo
console.log(num2);
num2=num2.toString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);

```

Red arrows point from the highlighted line (`num2=num2.toString();`) to the output lines. The first two output lines show the string representation of the number: "888.8888" and "string". The third output line shows the type: "number". The final output line shows the original number again: "888.8888".

Otras funciones útiles son:

- isNaN(valor):** nos dice si el valor pasado es un número (false), o no (true)
- isFinite(valor):** devuelve true si el valor es finito (no es *Infinity* ni *-Infinity*).
- parseInt(valor):** convierte el valor pasado a un número entero. Siempre que comience por un número la conversión se podrá hacer. Ej.:

```

parseInt(3.65)      // Devuelve 3
parseInt('3.65')    // Devuelve 3
parseInt('3 manzanas') // Devuelve 3, Number devolvería NaN

```

- parseFloat(valor):** como la anterior, pero conserva los decimales

OJO: al sumar números decimales (*floats*) podemos tener problemas:

```
console.log(0.1 + 0.2) // imprime 0.30000000000000004
```

Para evitarlo:

- Redondead los resultados.
- También se puede hacer una operación similar a $(0.1*10 + 0.2*10) / 10$.

console.log((0.1 + 0.2));	0.30000000000000004
console.log((0.1 + 0.2).toFixed(1));	0.3
console.log((0.1*10 + 0.2*10) / 10);	0.3

EJERCICIO: Modifica la función de calcular la nota media para que devuelva la media con 1 decimal

EJERCICIO: Modifica la función que devuelve el cubo de un número para que compruebe si el parámetro pasado es un número entero. Si no es un entero o no es un número mostrará un alert indicando cuál es el problema y devolverá false.

String

Las cadenas de texto van entre comillas simples o dobles, es indiferente. Podemos escapar un carácter con \ para poder usarlo dentro de la cadena. (Ejemplo: 'Hola \'Mundo\' ' devuelve Hola 'Mundo').

console.log("hola \"majo\""); console.log('Hola "majo"');	hola "majo" Hola "majo"
--	----------------------------

Para forzar la conversión a cadena se usa la función **String(valor)** (ej. `String(23)` devuelve '23')

El operador de concatenación de cadenas es `+`. Ojo porque si pedimos un dato con `prompt` siempre devuelve una cadena así que si le pedimos la edad al usuario (por ejemplo 20) y se sumamos 10 tendremos 2010 ('20'+10).

Algunos métodos y propiedades de las cadenas son:

- **`.length`**: devuelve la longitud de una cadena. Ej.: 'Hola mundo'.length devuelve 10
- **`.charAt(posición)`**: 'Hola mundo'.charAt(0) devuelve 'H'
- **`.indexOf(carácter)`**: 'Hola mundo'.indexOf('o') devuelve 1. Si no se encuentra devuelve -1
- **`.lastIndexOf(carácter)`**: 'Hola mundo'.lastIndexOf('o') devuelve 9
- **`.substring(desde, hasta)`**: 'Hola mundo'.substring(2,4) devuelve 'la'
- **`.substr(desde, num caracteres)`**: 'Hola mundo'.substr(2,4) devuelve 'la m'
- **`.replace(busco, reemplaza)`**: 'Hola mundo'.replace('Hola', 'Adiós') devuelve 'Adiós mundo'
- **`.toLocaleLowerCase()`**: 'Hola mundo'.toLocaleLowerCase() devuelve 'hola mundo'
- **`.toLocaleUpperCase()`**: 'Hola mundo'.toLocaleUpperCase() devuelve 'HOLA MUNDO'
- **`.localeCompare(cadena)`**: devuelve -1 si la cadena a que se aplica el método es anterior alfabéticamente a 'cadena', 1 si es posterior y 0 si ambas son iguales. Tiene en cuenta caracteres locales como acentos ñ, ç, etc
- **`.trim(cadena)`**: ' Hola mundo '.trim() devuelve 'Hola mundo'
- **`.startsWith(cadena)`**: 'Hola mundo'.startsWith('Hol') devuelve true
- **`.endsWith(cadena)`**: 'Hola mundo'.endsWith('Hol') devuelve false
- **`.includes(cadena)`**: 'Hola mundo'.includes('mun') devuelve true
- **`.repeat(veces)`**: 'Hola mundo'.repeat(3) devuelve 'Hola mundoHola mundoHola mundo'
- **`.split(separador)`**: 'Hola mundo'.split(' ') devuelve el array ['Hola', 'mundo']. 'Hola mundo'.split('') devuelve el array ['H', 'o', 'l', 'a', '', 'm', 'u', 'n', 'd', 'o']

Podemos probar los diferentes métodos en la página de [w3schools](#).

Template literals

Desde ES2015 también podemos poner una cadena entre ` (acento grave) y en ese caso podemos poner dentro variables y expresiones que serán evaluadas al ponerlas dentro de `${}`. También se respetan los saltos de línea, tabuladores, etc que haya dentro. Ejemplo:

```
let edad=25;
console.log(`El usuario tiene:
${edad} años`);
```

```
El usuario tiene:
25 años
```

Tratamiento de una cadena como un array

```

let cadena = "abcdefg";
console.log(`Cadena es el array: ${cadena}`);
console.log(`Valor en la posición 0: ${cadena[0]}`);
console.log(`Valor en la posición 1: ${cadena[1]}`);
console.log(`Valor en la posición última: ${cadena[cadena.length - 1]}`);
for (const letra of cadena) {
    console.log(letra);
}

```

The screenshot shows the browser's developer tools with the 'Console' tab selected. The code above is run, and the output is displayed in the console window. Red arrows point from the code to the output, highlighting specific parts of the code and the resulting values.

Cadena es el array: abcdefg
 Valor en la posición 0: a
 Valor en la posición 1: b
 Valor en la posición última: g
 a
 b
 c
 d
 e
 f
 g

EJERCICIO: Haz una función a la que se le pasa un DNI (ej. 12345678w o 87654321T) y devolverá si es correcto o no. La letra que debe corresponder a un DNI correcto se obtiene dividiendo la parte numérica entre 23 y cogiendo de la cadena 'TRWAGMYFPDXBNJZSQVHLCKE' la letra correspondiente al resto de la división. Por ejemplo, si el resto es 0 la letra será la T y si es 4 será la G. Prueba la función en la consola con tu DNI

Boolean

Los valores booleanos son **true** y **false**. Para convertir algo a booleano se usar **Boolean(valor)** aunque también puede hacerse con la doble negación (**!!**). Cualquier valor se evaluará a *true* excepto 0, NaN, null, undefined o una cadena vacía ("") que se evaluarán a *false*.

```

console.log(Boolean("pepe"));
console.log(!!("pepe"));
console.log(!("pepe"));
console.log(!null);
console.log (!!null);
console.log (!!" ");
console.log (!!"" );
console.log (!!undefined);

```

The screenshot shows the browser's developer tools with the 'Console' tab selected. The code above is run, and the output is displayed in the console window. Red arrows point from the code to the output, highlighting the conversion results.

true
 true
 false
 true
 false
 true
 false
 false
 false

Los operadores lógicos son ! (negación), && (and), || (or).

Para comparar valores tenemos == y ===. La triple igualdad devuelve *true* si son de igual valor y del mismo tipo. Como Javascript hace conversiones de tipos automáticas conviene usar la === para evitar cosas como:

- '3' == 3 true
- 3 == 3.0 true
- 0 == false true
- '' == false true
- ' ' == false true
- [] == false true
- null == false false
- undefined == false false
- undefined == null true

También tenemos 2 operadores lógicos para *diferente*: != y !== que se comportan como hemos dicho antes.

Los operadores relacionales son >, >=, <, <=. Cuando se compara un número y una cadena, ésta se convierte a número y no al revés (23 > '5' devuelve *true*, aunque '23' > '5' devuelve *false*)

console.log(23 > 5);	true
console.log('23' > '5');	false
console.log('23' > 5);	true
console.log(23 > '5');	true
.	.



DWEC – Javascript Web Cliente.

JavaScript 01 – Sintaxis (IV)

JavaScript 01 – Sintaxis (IV).....	1
Manejo de errores	1
Buenas prácticas	4
'use strict'	4
Variables	4
Otras	4
Clean Code.....	5

Manejo de errores

Si sucede un error en nuestro código el programa dejará de ejecutarse, por lo que el usuario tendrá la sensación de que no hace nada (el error sólo se muestra en la consola y el usuario no suele abrirla nunca). Para evitarlo debemos intentar capturar los posibles errores de nuestro código antes de que se produzcan.

Pero hay una construcción sintáctica **try...catch** que nos permite “atrapar” errores para que el script pueda, en lugar de morir, hacer algo más razonable.

```
try {  
    // código...  
} catch (err) {  
    // manipulación de error  
}
```

Dentro del bloque *try* ponemos el código que queremos proteger y cualquier error producido en él será pasado al bloque *catch* donde es tratado. Opcionalmente podemos tener al final un bloque *finally* que se ejecuta tanto si se produce un error como si no.

Funciona así:

- Primero, se ejecuta el código en *try { ... }*.
- Si no hubo errores, se ignora *catch (err)*: la ejecución llega al final de *try* y continúa, omitiendo *catch*.
- Si se produce un error, la ejecución de *try* se detiene y el control fluye al comienzo de *catch (err)*.
- La variable *err* (podemos usar cualquier nombre para ella) contendrá un objeto de error con detalles sobre lo que sucedió.

El parámetro que recibe *catch* es un objeto con las propiedades *name*, que indica el tipo de error (*SyntaxError*, *RangeError*, ... o el genérico *Error*), y *message*, que indica el texto del error producido.

```
try {
  jaja
} catch (err) {
  console.log(err);
  console.log(err.name);
  console.log(err.message);
}
```

ReferenceError: jaja is not defined
at funciones.js:10:3
ReferenceError
jaja is not defined

Para que `try..catch` funcione, el código debe ser ejecutable. En otras palabras, debería ser JavaScript válido.

No funcionará si el código es sintácticamente incorrecto, por ejemplo, si hay llaves sin cerrar.

El motor de JavaScript primero lee el código y luego lo ejecuta. Los errores que ocurren en la fase de lectura se denominan errores de “tiempo de análisis” y son irrecuperables (desde dentro de ese código). Eso es porque el motor no puede entender el código.

Entonces, `try...catch` solo puede manejar errores que ocurren en un código válido. Dichos errores se denominan “errores de tiempo de ejecución” o, a veces, “excepciones”.

En ocasiones podemos querer que nuestro código genere un error. Esto evita que tengamos que comprobar si el valor devuelto por una función es el adecuado o es un código de error.

Sin usar `try..catch`

```
1 jaja;
2 console.log("no sigue");
3
```

Uncaught ReferenceError: jaja is not defined
at funciones.js:1:1

En este caso no seguiría la ejecución (se para la aplicación) y la línea 2 no se ejecuta.

Usando `try..catch`

```
1 try {
  2   jaja;
3 } catch (error) {
4   console.error(error);
5 }
6 console.log("Ahora sí continúa la ejecución");
7
```

ReferenceError: jaja is not defined
at funciones.js:2:5
Ahora sí continúa la ejecución

Ejemplo de excepción definida por el programador:

Tenemos una función para retirar dinero de una cuenta que recibe:

- el saldo de la cuenta
- y la cantidad de dinero a retirar

La función devuelve:

- el nuevo saldo, pero si no hay suficiente saldo no debería restar nada sino mostrar un mensaje al usuario.

Si no gestionamos los errores haríamos:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    return false;
  }
  return saldo - cantidad;
```

```

}

let saldo = 30;
cantidad = 200;
let resultado = retirar(saldo, cantidad);

if (resultado === false) {
  console.log("Saldo insuficiente");
} else {
  saldo = resultado;
}

```

Se trata de un código poco claro que podemos mejorar lanzando un error en la función. Para ello se utiliza la instrucción `throw` que tiene la sintaxis: `throw expresión;`

```

if (saldo < cantidad) {
  throw 'Saldo insuficiente'
}

```

Por defecto al lanzar un error, este será de clase `Error` pero (el código anterior es equivalente a `throw new Error('Valor no válido')`) aunque podemos lanzarlo de cualquier otra clase (`throw new RangeError('Saldo insuficiente')`) o personalizarlo.

Siempre que vayamos a ejecutar código que pueda generar un error debemos ponerlo dentro de un bloque `try`.

Por lo que la llamada a la función que contiene el código anterior debería estar dentro de un `try`. El código del ejemplo anterior quedaría:

```

function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    // console.log("No hay saldo"); // esta línea no evita resultado erróneo
    throw "Saldo insuficiente";
  }
  return saldo - cantidad;
}

// Siempre debemos llamar a esa función desde un bloque _try_

let saldo = 30;
let importe = 200;

try {
  saldo = console.log(`Nuevo saldo: ${retirar(saldo, importe)}`);
} catch (err) {
  console.log(err); // muestra "Saldo Insuficiente"
}

try {
  saldo = console.log(`Nuevo saldo: ${retirar(200, 5)}`); //Muestra "Nuevo Saldo: 195"
} catch (err) {
  console.log(err);
}

```

Podemos ver en detalle cómo funcionan en la página de [MDN web docs](#) y <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/throw> de Mozilla.

Try...catch trabaja sincrónicamente

Si ocurre una excepción en el código “programado”, como en `setTimeout`, entonces `try..catch` no lo detectará.

Más información en: <https://es.javascript.info/try-catch>

EJERCICIO: Escribe una función que devuelva la nota media de un array de notas. Si el array está vacío debe avisar del error. Prueba la función con `try catch`:

Buenas prácticas

Javascript nos permite hacer muchas cosas que otros lenguajes no nos dejan, por lo que debemos ser cuidadosos para no cometer errores de los que **no se nos va a avisar**.

‘use strict’

Si ponemos siempre esta sentencia al principio de nuestro código el intérprete nos avisará si usamos una variable sin declarar (muchas veces por equivocarnos al escribir su nombre). En concreto fuerza al navegador a no permitir:

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal (comienzan por 0)
- Modificar una propiedad de sólo lectura

Variables

Algunas de las prácticas que deberíamos seguir respecto a las variables son:

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, ...)
- Evitar en lo posible variables globales
- Usar `let` para declararlas
- Usar `const` siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar, para nombrarlas, la notación `camelCase`

También es conveniente, por motivos de eficiencia no usar objetos `Number`, `String` o `Boolean`, si no los tipos primitivos: (no usar `let numero = new Number(5)`, sino `let numero = 5`) y lo mismo al crear arrays, objetos o expresiones regulares (no usar `let miArray = new Array()`, sino `let miArray = []`).

Otras

Algunas reglas más que deberíamos seguir son:

- Debemos ser coherentes a la hora de escribir código: por ejemplo, podemos poner (recomendado) o no espacios antes y después del `=` en una asignación, pero debemos hacerlo siempre igual. Existen muchas

guías de estilo y muy buenas: [Airbnb](#), [Google](#), [Idiomatic](#), etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta [linter](#).

- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres. Para saltar de línea en una cadena usamos el símbolo \

```
let cadena = "Esta es una cadena muy larga \
que me gustaría dividir en dos";
console.log (cadena);
//El resultado sería "Esta es una cadena muy larga que me gustaría dividir en dos";
```

- Usar === en las comparaciones
- Si un parámetro puede faltar al llamar a una función, darle un valor por defecto
- Y para acabar: **comentar el código** cuando sea necesario, pero mejor que el código sea lo suficientemente claro como para no necesitar comentarios

Clean Code

Estas y otras muchas recomendaciones se recogen en el libro [Clean Code](#) de Robert C. Martin y en muchos otros libros y artículos. Aquí se puede ver un pequeño resumen traducido al castellano:

<https://github.com/devictoribero/clean-code-javascript>



DWEC – Javascript Web Cliente.

JavaScript 02 – Arrays (I).....	2
Introducción.....	2
Objetos JavaScript.....	2
Propiedades de un objeto.....	2
Métodos de un objeto	3
Los arrays en JavaScript	4
Arrays de objetos	5
Operaciones con Arrays	5
length	5
Añadir elementos.....	5
Eliminar elementos	5
splice	6
slice	6
Arrays y Strings	6
sort.....	7
Otros métodos comunes.....	8
Functional Programming.....	9
filter.....	9
find	10
findIndex	11
every / some	11
map	11
reduce	11
forEach	14
includes	14
Array.from.....	14
Referencia vs Copia.....	15
Rest y Spread	16
Desestructuración de arrays	17
Los objetos Map y Set	18
Map.....	18
Set	19

JavaScript 02 – Arrays (I)

Introducción

Objetos JavaScript

Los objetos son uno de los tipos de datos de JavaScript.

Los objetos se utilizan para almacenar colecciones de clave/valor (nombre/valor).

Se puede decir que un objeto JavaScript es una colección de valores con nombre.

En Javascript podemos definir cualquier variable como un objeto declarándola con **new** (NO se recomienda) o creando un *literal object* (usando notación **JSON**).

El siguiente ejemplo crea un objeto JavaScript con tres propiedades clave/valor.

Creando con *new* (no recomendado):

```
let alumno = new Object()
alumno.nombre = 'Carlos'      // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz' // se crea la propiedad 'apellidos'
alumno.edad = 19
```

Creando un *literal object* (es la forma **recomendada**). El ejemplo anterior quedaría:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
};
```

Propiedades de un objeto

Podemos acceder a las propiedades con . (punto) o []:

```
console.log(alumno.nombre)      // imprime 'Carlos'
console.log(alumno['nombre'])    // imprime 'Carlos'
let prop = 'nombre'
console.log(alumno[prop])       // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen, no se produce un error: se devuelve *undefined*:

```
console.log(alumno.ciclo)        // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo)        // muestra undefined
console.log(alumno.ciclo.descrip) // se genera un ERROR
```

Para evitar ese error antes había que comprobar que existían las propiedades previas:

```
console.log(alumno.ciclo && alumno.ciclo.descrip)
```

```
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador `?.` para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip)
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Podremos recorrer las propiedades de un objeto con `for..in`:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

Nota: En este caso no se puede aplicar el bloque de código `for..of` porque un objeto de este tipo no es iterable, sí son iterables los arrays y strings

Si el valor de una propiedad es el valor de una variable que se llama como la propiedad, no es necesario ponerlo:

```
let nombre = 'Carlos'

let alumno = {
    nombre,           // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
}
```

Métodos de un objeto

Llamamos **método de un objeto** cuando una **propiedad** de ese objeto es una **función**:

```
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
}
```

Nota: La palabra clave `this` dentro de un método hace referencia al objeto en cuestión al que se aplica dicho método.

Y para llamarlo se hace como con cualquier otra propiedad:

```
console.log(alumno.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Ojo!! No se puede utilizar `this` con sintaxis `arrow function`. Está explicado en el documento **Anexo – Uso de this en contexto**

Algunos métodos y propiedades genéricas de objetos de JavaScript

- `constructor` Returns the function that created an object's prototype
- `keys()` Returns an Array Iterator object with the keys of an object
- `prototype` Let you to add properties and methods to JavaScript objects
- `toString()` Converts an object to a string and returns the result. Este método se usa sobreescrito.
- `valueOf()` Returns the primitive value of an object

Prueba las siguientes sentencias:

```
console.log(alumno.constructor);
console.log(Object.keys(alumno));
console.log(Object.values(alumno));
console.log(Object.entries(alumno));
console.log(alumno.toString());
alumno.edasd=33;
console.log(alumno.valueOf());
console.log(Object.prototype);
console.log(Object.constructor);
```

EJERCICIO: Crea un objeto llamado **tvSamsung** con las propiedades **nombre** ("TV Samsung 42"), **categoría** ("Televisores"), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (nº de unidades * precio).

Los arrays en JavaScript

Los arrays, también llamados arreglos, vectores, matrices, listas..., son un tipo de objeto que no tiene tamaño fijo, podemos añadirle elementos en cualquier momento.

Para hacer referencia a (referenciar) los elementos, se hace con un índice numérico. A diferencia de los objetos que se referencian con un nombre.

Podemos crearlos como instancias del objeto Array (No se recomienda):

```
let a = new Array()          // a = []
let b = new Array(2,4,6)    // b = [2, 4, 6]
```

Lo recomendable es crearlos usando notación JSON:

```
let a = []
let b = [2,4,6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array.

Si no está definido un elemento, su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0]) // imprime 'Lunes'
console.log(a[4]) // imprime 6
a[7] = 'Juan'     // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7]) // imprime 'Juan'
console.log(a[6]) // imprime undefined
```

Acceder a un elemento de un array que no existe no provoca un error (devuelve *undefined*), pero sí lo provoca acceder a un elemento de algo que no es un array. Con ES2020 (ES11) se ha incluido el operador **?** para evitar tener que comprobar nosotros que sea un array:

```
console.log(alumnos?[0])
// si alumnos es un array muestra el valor de su primer
// elemento y si no muestra undefined pero no lanza un error
```

Arrays de objetos

Es habitual almacenar datos en arrays en forma de objetos, por ejemplo:

```
let alumnos = [
  {
    id: 1,
    name: 'Carlos Pérez',
    course: '2DAW',
    age: 21
  },
  {
    id: 2,
    name: 'Ana García',
    course: '2DAW',
    age: 23
  },
];
```

Operaciones con Arrays

Los arrays tienen las mismas propiedades y métodos que los objetos, y muchos más que son propios de los arrays.

Vamos a ver los principales métodos y propiedades de los arrays.

length

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a.length) // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad:

```
a.length = 3 // ahora a = ['Lunes', 'Martes', 2]
```

Añadir elementos

Podemos añadir elementos al final de un array con el método `push`, o al principio con `unshift`:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
a.push('Juan') // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']
a.unshift(7) // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

Eliminar elementos

Podemos borrar el elemento del final de un array con `pop`, o del principio con `shift`. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let ultimo = a.pop() // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6
let primero = a.shift() // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3)           // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56)      // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56)      // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz lo siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos, Naranjas y Sandía. (Debe quedar: Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos. (Debe quedar: Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

slice

Devuelve un subarray con los elementos indicados, pero sin modificar el array original. Sería como hacer un substr pero de un array en vez de una cadena.

Sintaxis: `Array.slice(posicion, num. de elementos a devolver)`

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3) // ahora a=[ 'Lunes', 'Martes', 2, 4, 6] y subArray=['Martes', 2, 4]
```

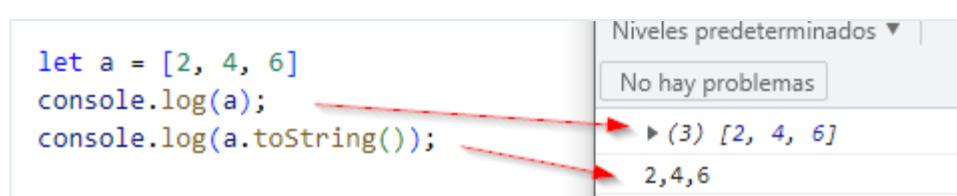
Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice() // ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

Arrays y Strings

Cada objeto, y los arrays son un tipo de objeto, tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola.

En el caso de los arrays, esta función devuelve una cadena con los elementos del array separados por coma.



Además, podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-') // cadena = 'Lunes-Martes-2-4-6'
```

El método `.join()` es el contrario del `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-') // arrayNotas = ['5', '3.9', '6', '9.75', '7.5', '3']
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ') // arrayPalabras = ['Que', 'tal', 'estás']
let arrayLetras = cadena.split('') // arrayLetras = ['Q','u','e`,' ','t',a',l',' ','e',s',t',á',s']
```

sort

Ordena **alfabéticamente** los elementos del array.

OJO!! El array original queda modificado con el nuevo orden.

```
let a = ['hola', 'adios', 'Bien', 'Mal', 2, 5, 13, 45]
let b = a.sort() // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

Si no se especifica otra cosa, el orden que se sigue es el de los códigos ascii, por lo que los dígitos numéricos van antes que las letras, y las mayúsculas antes que las minúsculas.

También podemos pasárle una función que le indique cómo ordenar. Esta función debe devolver un valor negativo si el primer elemento es menor, positivo si es mayor, o 0 si son iguales.

Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola', 'adios', 'Bien', 'Mal']
let b = a.sort(function(elem1, elem2) {
  if (elem1.toLocaleLowerCase() < elem2.toLocaleLowerCase()) return -1;
  if (elem1.toLocaleLowerCase() > elem2.toLocaleLowerCase()) return 1;
  if (elem1.toLocaleLowerCase() = elem2.toLocaleLowerCase()) return 0;
});
// b = ["adios", "Bien", "hola", "Mal"]
```

También se utiliza para ordenar números, tanto de forma ascendente como descendente:

```
let a=[20,4,87,2];
let b= a.sort(function(elem1,elem2){return elem1-elem2}) //ascendente
console.log(b);

b= a.sort(function(elem1,elem2){return elem2 - elem1}) //descendente
console.log(b);
```

Es frecuente utilizar esta función para ordenar arrays de objetos. Por ejemplo, si tenemos un objeto `persona` con los campos `nombre` y `edad`, para ordenar un array de objetos persona por su edad haremos:

```
let persona1 = {nombre: 'Juan', edad: 25};
```

```

let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

let personasOrdenado = personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
})

```

The screenshot shows a browser developer tools console with the following code and output:

```

let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
})
console.log(personas);

```

The output window shows:

- No hay problemas
- (3) [{} , {} , {}]
- 0: {nombre: 'Juan', edad: 25}
- 1: {nombre: 'Ana', edad: 33}
- 2: {nombre: 'Benito', edad: 52}
- length: 3
- [[Prototype]]: Array(0)

Usando arrow functions quedaría más sencillo:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.edad - persona2.edad)
```

Si lo que queremos es ordenar por un campo de texto podemos usar la función `toLocaleCompare`:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
```

The screenshot shows a browser developer tools console with the following code and output:

```

let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
console.log(personas);

```

The output window shows:

- No hay problemas
- (3) [{} , {} , {}]
- 0: {nombre: 'Ana', edad: 33}
- 1: {nombre: 'Benito', edad: 52}
- 2: {nombre: 'Juan', edad: 25}
- length: 3
- [[Prototype]]: Array(0)

EJERCICIO: Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10]. Pruébalo en la consola

Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

- `.concat()`: concatena arrays

```

let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b)      // c = [2, 4, 6, 'a', 'b', 'c']

```

- `.reverse()`: invierte el orden de los elementos del array

```

let a = [2, 4, 6]
let b = a.reverse()      // b = [6, 4, 2]

```

- `.indexOf()`: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array

- `.lastIndexOf()`: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array

```
let a = [2, 4, 6, 4]
console.log(a.indexOf(4));           // 1
console.log(a.lastIndexOf(4));      // 3
console.log(a.lastIndexOf('4'));    // -1
```

Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo.

El ejemplo más claro es que intenta evitar los bucles `for` y `while` sobre arrays o listas de elementos.

Normalmente, cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programming* es que a la función que debe hacer eso, además de pasarle como parámetro la lista sobre la que debe actuar, se le pasa como segundo parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 javascript incorpora métodos de *functional programming* en el lenguaje, especialmente para trabajar con arrays:

filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica.

Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array.

Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar).

Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo.

La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado, y **false** para el resto.

Ejemplo: dado un array con notas, devolver un array con las notas de los aprobados.

Usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0; i < arrayNotas.length; i++) {
  let nota = arrayNotas[i]
  if (nota >= 5) {
    aprobados.push(nota)
  }
} // aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota >= 5) {
    return true
  }
})
```

```

} else {
  return false
}
}) // aprobados = [5.2, 6, 9.75, 7.5]

```

Podemos refactorizar esta función para que sea más compacta:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  return nota >= 5 // nota >= 5 se evalúa a 'true' si es cierto, o 'false' si no lo es
})
// aprobados = [5.2, 6, 9.75, 7.5]

```

Y usando funciones tipo flecha la sintaxis queda mucho más simple:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(nota => nota >= 5)
// aprobados = [5.2, 6, 9.75, 7.5]

```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

EJERCICIO: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

find

Como *filter* pero NO devuelve un **array**, devuelve el primer **elemento** que cumpla la condición (o *undefined* si no la cumple ninguno). Ejemplo:

```

let arrayNotas = [4, 5.2, 3.9, 6, 9.75, 7.5, 3]
let primerAprobado = arrayNotas.find(nota => nota >= 5) // primerAprobado = 5.2

```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar un coche de color rojo dentro de un array llamado *coches* cuyos elementos son objetos con un campo 'color', haremos:

```

let coches = [
  {
    "color": "morado",
    "tipo": "berlina",
    "capacidad": 7
  },
  {
    "color": "rojo",
    "tipo": "camioneta",
    "capacidad": 5
  },
  {
    "color": "rojo",
    "tipo": "furgón",
    "capacidad": 7
  }
]

```

```
let cocheBuscado = coches.find(coche => coche.color === 'rojo') // devolverá el objeto completo del primer elemento que cumpla la condición.
```

Si queremos que nos devuelva el objeto en la posición “1” dentro del array, haremos:

```
let cocheBuscado=coches.find((coche,indice) => indice ===1); // devolverá el objeto coches[1]
```

EJERCICIO: Dado un array con los días de la semana obtén el primer día que empieza por ‘M’

findIndex

Como *find* pero, en vez de devolver el elemento, devuelve su posición (-1 si ningún elemento cumple la condición).

```
let cocheBuscado = coches.findIndex(coche => coche.color === 'rojo') // devolverá 1
```

En el ejemplo de los coches el valor devuelto sería 1, ya que el segundo elemento cumple la condición.

Al igual que el anterior, *findIndex* tiene más sentido con arrays de objetos.

EJERCICIO: Dado un array con los días de la semana, obtén la posición en el array del primer día que empieza por ‘M’.

every / some

- **every** devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario.
- **some** devuelve **true** si **ALGÚN** elemento del array cumple la condición. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota => nota >= 5) // false
let algunAprobado = arrayNotas.some(nota => nota >= 5) // true
```

EJERCICIO: Dado un array con los días de la semana indica si algún día empieza por ‘S’. Dado un array con los días de la semana indica si todos los días acaban por ‘s’

map

map permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados.

Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayNotasSubidas = arrayNotas.map(nota => nota + nota * 0.1);
```

EJERCICIO: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas.

reduce

El método **reduce**:

- Ejecuta una función tipo callback para cada elemento del array, y devuelve **un único valor calculado** a partir de los elementos del array, este valor es el resultado acumulado de las llamadas a la función.
- No se ejecuta la función para elementos vacíos del array.

- No se cambia el contenido del array original.

Sintaxis: `array.reduce(function(total, currentValue, currentIndex, vector), initialValue)`

Hay dos parámetros: una **función** e **initialValue** (este segundo parámetro es opcional).

La función admite 4 parámetros:

- **total o valorAnterior** (obligatorio), contiene el valor devuelto por la función en cada llamada.
- **currentValue** (obligatorio), contiene el valor del elemento actual.
- **currentIndex** (opcional), contiene el índice del elemento actual.
- **vector** (opcional), es el array al que pertenece el elemento actual.

La primera vez que se llama la función, `valorAnterior` y `valorActual` pueden tener uno de dos posibles valores:

- Si se suministró un `valorInicial` al llamar a `reduce`, entonces `valorAnterior` será igual al `valorInicial` y `valorActual` será igual al primer elemento del array.
- Si no se proporcionó un `valorInicial`, entonces `valorAnterior` será igual al primer valor en el array y `valorActual` será el segundo.

Ejemplo: queremos obtener la suma de las notas de un array:

```
let arrayNotas = [4,7,5];
let suma=0;

suma= arrayNotas.reduce((total, valor) => total+= valor, 0);
console.log(suma); // obtiene 16

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor) => {
    return total+= valor
}, 0);
console.log(suma); // obtiene 16
```

La función segunda permite añadir fácilmente otras funcionalidades

Para observar cómo funciona `reduce()`, estudia el siguiente caso para sumar el array de notas. Se ha añadido el parámetro del índice a la función callback.

Si añadimos un `valorInicial`, hace un recorrido por el índice cero. Si este valor es cero, el resultado es el mismo que sin no ponemos `valorInicial`.

<pre>let arrayNotas = [4,7,5]; let suma=0; // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) => { console.log('Valor: ' + valor); console.log('Índice: ' + i); return total+= valor; }, 0); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Niveles predeterminados No hay problemas</p> <table border="1" style="margin-top: 10px; width: 100px;"> <tr><td>Valor: 4</td></tr> <tr><td>Índice: 0</td></tr> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 4	Índice: 0	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16	<pre>let arrayNotas = [4,7,5]; let suma=0; // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) => { console.log('Valor: ' + valor); console.log('Índice: ' + i); return total+= valor; },); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Filtrar Niveles predeterminado No hay problemas</p> <table border="1" style="margin-top: 10px; width: 100px;"> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16
Valor: 4															
Índice: 0															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															

En el siguiente ejemplo hay dos casos. Entre el caso de la izda y el de la derecha solo cambia el valor del parámetro opcional `initialValue` que en este caso vale 3, por lo que el resultado cambia.

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
},);
console.log('Resultado: ' + suma); // obtiene 16 >
```

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
}, 3); ←
console.log('Resultado: ' + suma); // obtiene 19
```

Otros ejemplos similares de suma de notas:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let sumaNotas = arrayNotas.reduce((total,nota) => total += nota, 30) // total = 65.35
sumaNotas = arrayNotas.reduce((total,nota) => total += nota) // total = 35.35
```

Podemos tener la función declarada y utilizarla con reduce(). Ejemplo:

```

function suma(a, b) {
  return a + b;
}

const numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90];
const resultado = numeros.reduce(suma);
console.log(resultado); // 450
```

Otro ejemplo: queremos obtener la nota más alta:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota) => nota > max ? nota : max) // max = 9.75
```

Ejemplo muy útil: Integrar un array a partir de varios arrays utilizando el método array.concat()

```

let integrado = [[0,1], [2,3], [4,5]].reduce(function(a,b) {
  return a.concat(b);
});
// integrado es [0, 1, 2, 3, 4, 5]
```

Ejemplo clarificador de la idea: Cadena de montaje, se van añadiendo piezas a la cadena de texto inicial.

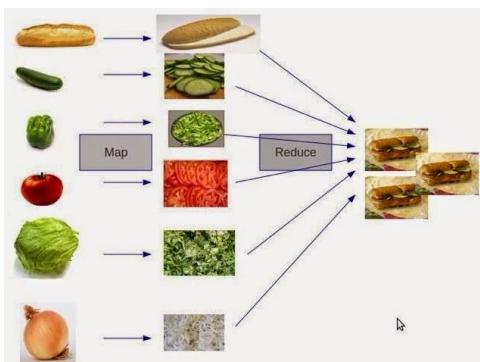
```

const partesDelCoche = ["asientos", "volante", "puertas", "ruedas", "pintura metalizada"];

const coche = partesDelCoche.reduce(function (valorAnterior, valorActual) {
  return `${valorAnterior} ${valorActual},`;
}, "Mi coche tiene: ");

console.log(coche);
//Mi coche tiene: asientos, volante, puertas, ruedas, pintura metalizada,
```

Una idea del funcionamiento de map y reduce sería: Tenemos un “array” de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sandwich) con todas ellas.



EJERCICIO: Dado el array de notas visto anteriormente, usar un método para que devuelva la nota media.

forEach

Es el método más general de los que hemos visto. No devuelve nada, sino que permite realizar algo con cada elemento del array.

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.forEach((nota, indice, arrayCompleto) => {
  console.log('El elemento de la posición ' + indice + ' es: ' + nota)
})
  
```

Los 3 argumentos de la función son: Valor de cada elemento del array, índice del array, y contenido completo del array.

includes

Devuelve **true** si el array incluye el elemento pasado como parámetro. Ejemplo:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.includes(7.5)      // true
  
```

EJERCICIO: Dado un array con los días de la semana indica si algún día es el 'Martes'

Array.from

Devuelve un array a partir de otro al que se puede aplicar una función de transformación (es similar a *map*). Ejemplo: queremos subir un 10% cada nota:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = Array.from(arrayNotas, nota => nota * 1.1)
  
```

Puede usarse para hacer una copia de un array, igual que *slice*:

```

let arrayA = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayB = Array.from(arrayA);
  
```

También se utiliza mucho para convertir colecciones en arrays y así poder usar los métodos de arrays que hemos visto con las colecciones.

Por ejemplo, si queremos mostrar por consola cada párrafo de la página que comience por la palabra 'Si' en primer lugar obtenemos todos los párrafos con:

```
let parrafos = document.getElementsByTagName('p')
```

Esto nos devuelve una colección con todos los párrafos de la página (lo veremos más adelante al ver DOM). Podríamos hacer un **for** para recorrer la colección y mirar los que empiecen por lo indicado, pero no podemos aplicarle los métodos vistos aquí porque son sólo para arrays, así que hacemos:

```
let parrafos = document.getElementsByTagName('p');
let arrayParrafos = Array.from(parrafos);
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo => parrafo.textContent.startsWith('Si'))
.forEach(parrafo => alert(parrafo.textContent))
```

IMPORTANTE: desde este momento se han acabado los bucles *for* en nuestro código para trabajar con arrays. Usaremos siempre estas funciones!!!

Existen otros métodos para utilizar con arrays, se pueden ver en:

https://www.w3schools.com/jsref/jsref_obj_array.asp

Referencia vs Copia

Cuando copiamos una variable de tipo *boolean*, *string* o *number* (o se pasa esa variable como parámetro a una función), se hace una copia de la misma. Por lo que, si se modifica, la variable original no es modificada. Ej.:

```
let a = 54
let b = a      // a = 54 b = 54
b = 86        // a = 54 b = 86
```

Sin embargo, al copiar objetos (y los arrays son un tipo de objeto), la nueva variable apunta a la misma posición de memoria que la antigua por lo que los datos de ambas son los mismos:

```
let a = [54, 23, 12]
let b = a      // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3      // a = [3, 23, 12] b = [3, 23, 12]
let fecha1 = new Date('2022-09-23')
let fecha2 = fecha1      // fecha1 = '2022-09-23'    fecha2 = '2022-09-23'
fecha2.setFullYear(1999) // fecha1 = '1999-09-23'    fecha2 = '1999-09-23'
```

Si queremos obtener una copia de un array que sea independiente del original podemos obtenerla con *slice* o con *Array.from*:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()  // ahora ambos arrays contienen lo mismo, pero son diferentes
let otraCopiaDeA = Array.from(a)
```

En el caso de objetos, es algo más complejo. ES6 incluye **Object.assign**, que hace una copia de un objeto:

```
let a = {id:2, name: 'object 2'}
let copiaDeA = Object.assign({}, a) //ahora ambos objetos contienen lo mismo, pero son diferentes
```

Sin embargo, si el objeto tiene como propiedades otros objetos, éstos se continúan pasando por referencia. En ese caso lo más sencillo sería utilizar *objetoCopia = JSON.stringify(objetoOriginal);*

JSON es un formato de texto para almacenar y transportar datos.

JavaScript tiene una función integrada para convertir cadenas JSON en objetos JavaScript: `JSON.parse()`

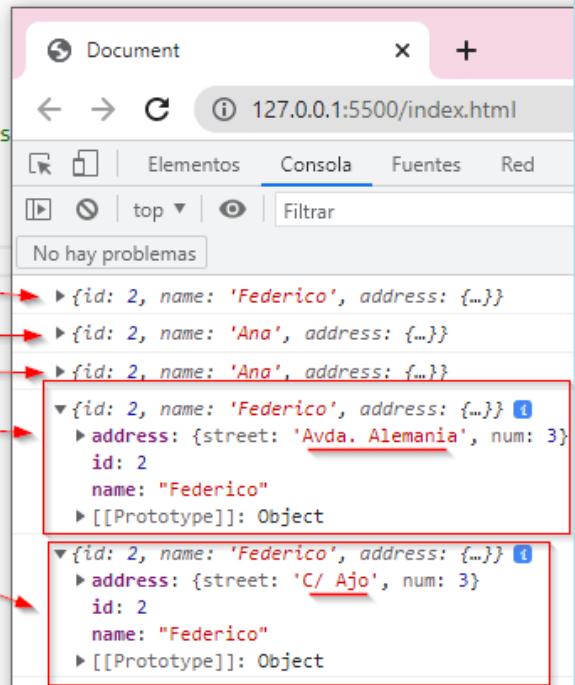
JavaScript tiene una función integrada para convertir un objeto en una cadena JSON: `JSON.stringify()`

```
let a = {id: 2, name: 'object 2', address: {street: 'C/ Ajo', num: 3} }
let copiaDeA = JSON.parse(JSON.stringify(a)) // ahora ambos objetos contienen lo mismo pero
son diferentes
```

```
'use strict';
let a = {id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3} }
let b = a;
let copia1= Object.assign({}, a);

//solución:
// ahora ambos objetos contienen lo mismo pero son diferentes
let copiaDeA = JSON.parse(JSON.stringify(a))

console.log((b));
a.name= 'Ana';
a.address.street = 'Avda. Alemania'
console.log(a);
console.log(b);
console.log(copia1);
console.log(copiaDeA);
```



EJERCICIO: Dado el array arr1 con los días de la semana haz un array arr2 que sea igual al arr1. Elimina de arr2 el último día y comprueba qué ha pasado con arr1. Repita la operación con un array llamado arr3 pero que crearás haciendo una copia de arr1.

Rest y Spread

También podemos copiar objetos usando *rest* y *spread*.

Permiten extraer a parámetros los elementos de un array o string (*spread*) o convertir en un array un grupo de parámetros (*rest*). El operador de ambos es ... (3 puntos).

Rest se utiliza para convertir en un array un grupo de parámetros (*rest*). El operador es ... (3 puntos).

Para usar *rest* como parámetro de una función debe ser siempre el último parámetro:

Ejemplo: queremos hacer una función que calcule la media de las notas que se le pasen como parámetro y que no sabemos cuántas son. Para llamar a la función haremos:

```
console.log(notaMedia(3.6, 6.8))
console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3))
```

La función notaMedia convertirá los parámetros recibidos en un array usando *rest*:

```
function notaMedia(...notas) {
```

```
let total = notas.reduce((total,nota) => total += nota)
return total/notas.length
}
```

Spread permite pasar como parámetros independientes los elementos de un array o string. El operador también es ... (3 puntos).

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*.

```
let array = [1,2,3];
console.log(array);
// hacemos la conversión de array a parámetros
console.log(...array);
```

Por ejemplo, el objeto *Math* proporciona métodos para trabajar con números como *.max* que devuelve el máximo de los números pasados como parámetro. Para saber la nota máxima en vez de utilizar *.reduce* como hicimos en el ejemplo anterior, podemos hacer:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let maximo = Math.max(arrayNotas); //maximo = NaN
// si hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número

// hacemos la conversión de array a parámetros
let maximoSpread = Math.max(...arrayNotas); // maxNota = 9.75
```

Estas funcionalidades nos ofrecen otra manera de copiar objetos (pero sólo a partir de ES-2018):

```
let a = {id: 2, name: 'object 2'}
let copiaDeA = { ...a} // ahora ambos objetos contienen lo mismo pero son diferentes
```

```
let b = [2, 8, 4, 6]
let copiaDeB = [ ...b ] // ahora ambos objetos contienen lo mismo pero son diferentes
```

Desestructuración de arrays

Similar a *rest* y *spread*, permiten extraer los elementos del array directamente a variables y viceversa. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let [primera, segunda, tercera] = arrayNotas // primera = 5.2, segunda = 3.9, tercera = 6
let [primera, , , cuarta] = arrayNotas // primera = 5.2, cuarta = 9.75
let [primera, ...resto] = arrayNotas // primera = 5.2, resto = [3.9, 6, 9.75, 3]
```

También se pueden asignar valores por defecto:

```
let preferencias = ['Javascript', 'NodeJS']
let [lenguaje, backend = 'Laravel', frontend = 'VueJS'] = preferencias
    // lenguaje = 'Javascript', backend = 'NodeJS', frontend = 'VueJS'
```

La desestructuración también funciona con objetos. Es normal pasar un objeto como parámetro para una función, pero si sólo nos interesan algunas propiedades del mismo, podemos desestructurarlo:

```
const miProducto = {
  id: 5,
  name: 'TV Samsung',
  units: 3,
  price: 395.95
};

// Se puede abreviar: function muestraNombre({name, units}) {
function muestraNombre({name: name, units: units}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre(miProducto); //Del producto TV Samsung hay 3 unidades
```

También podemos asignar valores por defecto:

```
function muestraNombre({name, units = 0}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre({name: 'USB Kingston'});
// mostraría: Del producto USB Kingston hay 0 unidades
```

Los objetos Map y Set

En el ámbito de JavaScript, a menudo los desarrolladores pasan mucho tiempo decidiendo la estructura de datos correcta que se usará. Esto se debe a que elegir la estructura de datos correcta puede facilitar la manipulación de esos datos posteriormente, con lo cual se puede ahorrar tiempo y facilitar la comprensión del código.

Las dos estructuras de datos predominantes para almacenar conjuntos de datos son los Object (Objetos) y Array (un tipo de objeto).

Los desarrolladores utilizan Object para almacenar pares clave-valor y Array para almacenar listas indexadas.

Sin embargo, para dar más flexibilidad a los desarrolladores, en la especificación ECMAScript 2015 se introdujeron dos nuevos tipos de objetos iterables:

- los Map, que son grupos ordenados de pares clave-valor.
- los Set, que son grupos de valores únicos.

Map

El objeto **Map** es una colección de parejas de [clave,valor].

En cambio, un objeto en Javascript es un tipo particular de *Map* en el que las claves sólo pueden ser texto o números.

Se puede acceder a una propiedad con `.` o **[propiedad]**. Ejemplo:

```
let persona = {
  nombre: 'John',
  apellido: 'Doe',
```

```

    edad: 39
}
console.log(persona.nombre)      // John
console.log(persona['nombre'])   // John

```

Un *Map* permite que la clave sea cualquier cosa (array, objeto, ...).

Más información en [MDN](#) o cualquier otra página.

Set

El objeto **Set** es como un *Map*, pero que no almacena los valores, almacena solo la clave. Podemos verlo como una colección que no permite duplicados.

Tiene la propiedad **size** que devuelve su tamaño y los métodos **.add** (añade un elemento), **.delete** (lo elimina) o **.has** (indica si el elemento pasado se encuentra o no en la colección) y también podemos recorrerlo con **.forEach**.

Una forma sencilla de eliminar los duplicados de un array es crear con él un *Set*:

```

let ganadores = ['Márquez', 'Rossi', 'Márquez', 'Lorenzo', 'Rossi', 'Márquez',
'Márquez']
let ganadoresNoDuplicados = new Set(ganadores)      // {'Márquez', 'Rossi', 'Lorenzo'}

// o si lo queremos en un array:

ganadoresNoDuplicados = Array.from(new Set(ganadores)) // ['Márquez', 'Rossi', 'Lorenzo']

```



DWEC – Javascript Web Cliente.

JavaScript 03 – Objetos y Clases en Javascript.....	1
Introducción.....	1
Propiedades de un objeto.....	2
Agregar y eliminar propiedades en un objeto:.....	4
Métodos get y set en objetos	4
Clases en Javascript.....	4
La palabra reservada this. Cuidado.	5
Herencia, sobreescritura, polimorfismo.....	6
Métodos estáticos.....	8
Atributos estáticos	9
Constantes estáticas	10
toString()	11
valueOf()	14
Métodos get y set en JS	14
Agregar propiedades y métodos a los objetos de una clase (prototype).....	15
Prototipos y POO en JS5.....	16
Uso de call y apply en Javascript.....	16
La clase Object	17

JavaScript 03 – Objetos y Clases en Javascript

Introducción

En Javascript podemos definir cualquier variable como un objeto, existen dos formas para hacerlo:

- Declarándola con **new** (NO se recomienda)
- Forma recomendada: creando un *literal object* usando notación **JSON**.

Ejemplo con *new*:

```
let alumno = new Object();
alumno.nombre = 'Carlos';      // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz';    // se crea la propiedad 'apellidos'
alumno.edad = 19;
```

Creando un *literal object* según la forma recomendada, el ejemplo anterior sería:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
```

```
edad: 19,  
};
```

Propiedades de un objeto

Podemos acceder a las propiedades con . (punto) o []:

```
console.log(alumno.nombre);           // imprime 'Carlos'  
console.log(alumno['nombre']);       // imprime 'Carlos'  
  
let prop = 'nombre';  
console.log(alumno[prop]);          // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen no se produce un error, se devuelve *undefined*:

```
console.log(alumno.ciclo);          // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo);          // muestra undefined  
console.log(alumno.ciclo.descrip);  // se genera un ERROR
```

En versiones anteriores de JavaScript, para evitar este error se comprobaba que existían las propiedades previamente. Veamos un ejemplo:

```
console.log(alumno.ciclo && alumno.ciclo.descrip);  
// si alumno.ciclo es un objeto muestra el valor de  
// alumno.ciclo.descrip y si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador ?. para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip);  
// si alumno.ciclo es un objeto muestra el valor de  
// alumno.ciclo.descrip y si no muestra undefined
```

Este nuevo operador también puede aplicarse a **arrays**:

```
let alumnos = ['Juan', 'Ana'];  
console.log(alumnos?.[0]);  
// si alumnos es un array y existe el primer elemento muestra el valor  
// si ese elemento no existe muestra undefined  
// si no existe el objeto con el nombre alumnos da ERROR
```

Podremos recorrer las propiedades de un objeto con `for..in`:

```
for (let prop in alumno) {  
  console.log(prop + ': ' + alumno[prop])  
}
```

Resultado:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

nombre: Carlos
apellidos: Pérez Ortiz
edad: 19

>

Una propiedad de un objeto puede ser una función:

```
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad +
    ' años'
}

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

También se puede incluir la declaración del método en la declaración del objeto:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: function(){
        return 'El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años';
    }
};

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

OJO: deberíamos poder ponerlo con sintaxis *arrow function*, pero no funciona.

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno undefined undefined tiene undefined años
```

No funciona bien porque *this* tiene distinto valor dependiendo del contexto, y no se puede usar en estos casos con función flecha. Tienes un documento titulado “JavaScript - Anexo - Uso de *this* en contexto” que lo explica.

Si el valor de una propiedad es el valor de una variable que se llama como ella, desde ES2015 no es necesario ponerlo:

```
let nombre = 'Carlos'
let alumno = {
    nombre,           // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
}
```

EJERCICIO: Crea un objeto llamado tvSamsung con las propiedades **nombre** (TV Samsung 42’), **categoria** (Televisores), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (nº de unidades * precio).

Prueba el uso del método con un ejemplo.

Agregar y eliminar propiedades en un objeto:

Se pueden agregar propiedades sin más. Pero OJO!! Si nos equivocamos al modificar el valor de una propiedad (escribimos un nombre de propiedad distinto) nos va a crear una propiedad que no deseamos.

Para eliminar una propiedad se hace con la palabra reservada **delete**

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
};

//añadimos la propiedad email
alumno.email='cperez@email.com';
console.log(alumno.email); //cperez@email.com

//eliminamos la proiedad email
delete alumno.email;
console.log(alumno); // {nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
```

Métodos get y set en objetos

Al igual que en las clases (como veremos más adelante), se pueden declarar métodos get y set en una o en varias propiedades de un objeto para acceder a su contenido o para modificar su valor.

Clases en Javascript

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes: clases, herencia, cohesión, abstracción, polimorfismo, acoplamiento, encapsulamiento... En Javascript solo se permite un método constructor.

Más información en https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

Veamos un ejemplo:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
  }
  getInfo() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
  }
}

let alumno1 = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(alumno1.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Recuerda que para las funciones y variables tipo var, da igual el orden (declaración primero y llamada después), o viceversa, pero para las clases no. Primero se declara la clase y luego se crean objetos.

Para las clases no se aplica el concepto de *hoisting*, por lo que no es posible crear objetos antes de declarar la clase.

EJERCICIO: Crea una clase Productos con las propiedades y métodos del ejercicio anterior (el de la TV). Además tendrá un método **getInfo** que devolverá: ‘Nombre (categoría): unidades uds x precio € = importe €’. Crea 3 productos diferentes y prueba getInfo.

La palabra reservada this. Cuidado.

Dentro de una función se crea un nuevo contexto y la variable *this* pasa a hacer referencia a dicho contexto. Si en el ejemplo anterior hiciéramos algo como esto:

```
class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    getInfo() {
        function nomAlum() {
            return this.nombre + ' ' + this.apellidos; // Aquí this no es el objeto Alumno
        }
        return 'El alumno' + nomAlum() + ' tiene ' + this.edad + ' años';
    }
}
```

Este código fallaría porque dentro de *nomAlum* la variable *this* ya no hace referencia al objeto *Alumno* sino al contexto de la función. Este ejemplo no tiene mucho sentido, pero a veces nos pasará en manejadores de eventos.

Si debemos llamar a una función dentro de un método (o de un manejador de eventos) tenemos varias formas de pasarle el valor de *this*:

1^a forma: Usando una *arrow function* que no crea un nuevo contexto, por lo que *this* conserva su valor.

```
getInfo() {
    let nomAlum=() => this.nombre + ' ' + this.apellidos;
    return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años';
}
```

2^a forma: Pasándole *this* como parámetro a la función:

```
getInfo() {
    function nomAlum(alumno){
        return alumno.nombre + ' ' + alumno.apellidos;
    }
    return 'El alumno ' + nomAlum(this) + ' tiene ' + this.edad + ' años';
}
```

3^a forma: Guardando el valor de *this* en otra variable (como *that*)

```
getInfo() {
    let that=this;
```

```

function nomAlum() {
    return that.nombre + " " + that.apellidos;
}
return "El alumno " + nomAlum() + " tiene " + this.edad + " años";
}

```

4^a forma: Haciendo un *bind* de *this*, se explica en el apartado de eventos.

Herencia, sobreescritura, polimorfismo

Nota: Las relaciones entre las clases se dan por **Herencia** (Especialización o Generalización), **Asociación**, **Agregación** (por Referencia), **Composición** (por Valor) y **Dependencia** (Uso). Más información en Anexo - Introducción a UML en el apartado “2.5 Diagrama de Clases”.

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos.

Los métodos podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** (que es lo que haremos si creamos un constructor en la clase hija)).

```

class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    getInfo() {
        return (
            "El alumno " + this.nombre + " " + this.apellidos + " tiene " + this.edad + " años"
        );
    }
}

class AlumnInf extends Alumno {
    constructor(nombre, apellidos, edad, ciclo) {
        super(nombre, apellidos, edad);
        this.ciclo = ciclo;
    }
    getGradoMedio() {
        if (this.ciclo.toUpperCase() === "SMR") return true;
        else return false;
    }
    getInfo() {
        return (
            super.getInfo() + " y estudia el Grado " +
            (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
    }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
console.log(azp.getInfo()); // imprime: 'El alumno Ana Zubiri Peláez tiene 24 años'

let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");

```

```
console.log(cpo.getInfo()); // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años y
estudia el Grado Superior de DAW'
```

En el ejemplo anterior hemos sobreescrito el método `getInfo()`, lo que implica que hay dos versiones de este método (la primera en la clase padre y la segunda en la clase hija). Se aplicará la primera versión de `getInfo()` o la segunda, dependiendo de si el objeto en cuestión es de una clase o de otra.

Siguiendo con el ejemplo visto anteriormente:

- Quitamos la salida por consola de ambos objetos
- Declaramos una función **imprimir()** fuera de las dos clases.
- Llamamos a la función imprimir pasando como argumento cada objeto

Al llamar a imprimir pasando como argumento cada objeto (cada uno es de un tipo), conseguimos que se llame, a su vez, al método `getInfo()` de la clase de ese objeto.

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  getInfo() {
    return (
      "El alumno " + this.nombre + " " + this.apellidos + " tiene " + this.edad + " años"
    );
  }
}

class AlumnInf extends Alumno {
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad);
    this.ciclo = ciclo;
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase() === "SMR") return true;
    else return false;
  }
  getInfo() {
    return (
      super.getInfo() + " y estudia el Grado " +
      (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
  }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");

function imprimir(objeto){
  console.log(objeto.getInfo());
}
```

```
imprimir(azp);
imprimir(cpo);
```

Es lo que se llama **polimorfismo**, pues hay múltiples formas en tiempo de ejecución, y se ejecutará un método u otro dependiendo de la referencia pasada como argumento.

InstanceOf

Para saber el tipo con el que trabajamos se utiliza **instanceof**

Así sabremos el tipo de objeto que estamos recibiendo.

```
function imprimir(objeto){
    console.log(objeto.getInfo());
    if (objeto instanceof AlumnInf){
        console.log(objeto.nombre + ' es un alumno de informática');
    }
}
```

En el ejemplo anterior, preguntamos si el objeto pasado como argumento es un tipo de la clase **AlumnInf**, de ser así, muestra por consola la frase “Fulanito es un alumno de informática”.

Hay que tener en cuenta que los objetos de la clase **AlumnInf** son también de la clase **Alumno** y también de la clase **Object**. Así que, para preguntar por el tipo de un parámetro, se suele hacer empezando por el tipo de menor jerarquía (el hijo).

EJERCICIO: crea una clase Televisores que hereda de Productos y que tiene una nueva propiedad llamada **tamaño**. El método **getInfo** mostrará el tamaño junto al nombre.

Métodos estáticos

Desde ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente **utilizando el nombre de la clase** y no tienen acceso al objeto **this** (ya que no hay objeto instanciado).

```
class User {
    static getRoles() {
        return ["user", "guest", "admin"]
    }
}

console.log(User.getRoles()) // ["user", "guest", "admin"]
let usuario = new User("john")
console.log(usuario.getRoles()) // Uncaught TypeError: usuario.getRoles is not a function
```

El siguiente ejemplo demuestra varias cosas:

1. Cómo se implementa método estático en una clase,
2. Que una clase con un miembro estático puede ser sub-claseada.
3. Finalmente demuestra cómo un método estático puede (y cómo no) ser llamado.

```
class Tripple {
    static tripple(n) {
        n = n || 1;
        return n * 3;
```

```

        }
    }

    class BiggerTripple extends Tripple {
        static tripple(n) {
            return super.tripple(n) * super.tripple(n);
        }
    }

    console.log(Tripple.tripple());           // 3
    console.log(Tripple.tripple(3));          // 9
    console.log(BiggerTripple.tripple(3));    // 81
    let tp = new Tripple();
    console.log(tp.tripple()); //ERROR Logs 'tp.tripple is not a function'.

```

Atributos estáticos

Se pueden definir atributos estáticos con la palabra reservada **static**, y será un atributo que pertenece a la clase y no al objeto.

Para hacer referencia (utilizar este atributo) hay que poner *NombreDeClase.atributo*

Es muy útil para llevar un contador de objetos de la clase, o para llevar un atributo de clave única autoincremenetal (que en nuestro caso hemos llamado *id*).

```

class Alumno {
    static contadorObjetosAlumno = 0;
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                    // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos.push(new Alumno('Ana', 'Zubiri Peláez', 29));

```

▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}

Este atributo contador también se hereda a las clases hijas de Alumno si las hubiere. Cada vez que se crea un objeto de una clase hija, incrementa este contador.

Resumiendo: Hemos creado una variable que tiene un identificador único para cada objeto que se crea, utilizando una variable estática.

Debido a que la variable estática se asocia con la clase y no con los objetos, permite asignar un valor único. Que en el ejemplo va preincrementándose para lograr que el id sea único y consecutivo, independientemente de que sea un objeto de la clase padre (donde está definido) o en la clase que hereda.

Constantes estáticas utilizando métodos estáticos

Para disponer de una variable estática de solo lectura, no podemos utilizar la palabra reservada **const**.

Lo que haremos es crear un método estático que solo permitirá leer el valor que devuelve el método, sin poder modificar este valor. Así parecerá que es una constante estática.

En el ejemplo siguiente declararemos la constante MAX_OBJ con el valor 5. (el máximo número de objetos que podremos crear serán 5, tanto de la clase padre como de la hija).

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                    // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

console.log(Alumno.MAX_OBJ); // 5
Alumno.MAX_OBJ = 10;
console.log(Alumno.MAX_OBJ); // 5
```

Como hemos definido un método **get**, podemos recuperar esta información de la forma **Alumno.MAX_OBJ**

Como no hemos definido método **set** para este atributo, no podremos modificar el valor.

Aunque no hemos creado una variable, este método funciona como si fuera una variable (constante).

Funciona igual que un atributo de la clase.

Siguiendo con el ejemplo: cambiamos el constructor:

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        if(Alumno.contadorObjetosAlumno < Alumno.MAX_OBJ){
            this.id = ++ Alumno.contadorObjetosAlumno;
        } else {
```

```

        console.log('Se ha superado el máximo de objetos permitidos de la clase
Alumno');
    }
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
    console.log(this);
}
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[3] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[4] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[5] = new Alumno('Ana', 'Zubiri Peláez', 29);

```

Importante a tener en cuenta: En el caso anterior se crearán 6 alumnos, el sexto alumno se ha creado porque no hemos mandado llamar a una excepción, pero no tiene atributo id porque así lo hemos decidido en el constructor:

```

▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
▶ Alumno {id: 4, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 5, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
Se ha superado el máximo de objetos permitidos de la clase Alumno
▶ Alumno {nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
▶ misAlumnos
▶ ▶ (6) [Alumno, Alumno, Alumno, Alumno, Alumno, Alumno]

```

toString()

De forma implícita, todas las clases que vayamos a crear heredan (*extends*) de la clase *Object*. En Javascript, la clase *Object* es la clase padre de todas las clases, por tanto, podemos usar el método heredado *toString()* o sobrescribirlo.

Al convertir un objeto a string (por ejemplo, al concatenarlo con un String) se llama al método ***.toString()*** del propio objeto, que por defecto devuelve la cadena [object Object]. Podemos sobrecargar este método para que devuelva lo que queramos:

```

class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }

  toString() {

```

```

        return this.apellidos + ', ' + this.nombre
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
console.log('Alumno:' + cpo)      // imprime 'Alumno: Pérez Ortiz, Carlos'
                                  // en vez de 'Alumno: [object Object]'
```

Este método también es el que se usará si queremos ordenar una array de objetos (recuerda que `.sort()` ordena alfabéticamente para lo que llama al método `.toString()` del objeto a ordenar).

Por ejemplo, tenemos el array de alumnos `misAlumnos` que queremos ordenar alfabéticamente por apellidos. Si la clase `Alumno` no tiene un método `toString` habría que hacer como vimos en el tema de Arrays:

```

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
    if (alum1.apellidos > alum2.apellidos) return 1
    if (alum1.apellidos < alum2.apellidos) return -1
});
```

NOTA: como las cadenas a comparar pueden tener acentos u otros caracteres propios del idioma, el código anterior no siempre funcionará bien. La forma correcta de comparar cadenas es usando el método `.localeCompare()`. El código anterior debería ser:

```

misAlumnos.sort(function(alum1, alum2) {
    return alum1.apellidos.localeCompare(alum2.apellidos)
});
```

que con *arrow function* quedaría:

```
misAlumnos.sort((alum1, alum2) => alum1.apellidos.localeCompare(alum2.apellidos) )
```

o si queremos comparar por 2 campos ('apellidos' y 'nombre')

```

misAlumnos.sort((alum1, alum2) =>
(alum1.apellidos+alum1.nombre).localeCompare(alum2.apellidos+alum2.nombre) )
```

Si sobreescrivimos el método `toString`, podemos utilizar este método para la ordenación:

```

class Alumno {
constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
}

toString() {
    return this.apellidos + ', ' + this.nombre
}
}
```

```
let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
  if (alum1.toString() > alum2.toString()) return 1;
  if (alum1.toString() < alum2.toString()) return -1;
});
```

Pero con el método `toString` que hemos definido antes podemos hacer directamente:

```
misAlumnos.sort()
```

Quedando el ejemplo al completo:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }

  toString() {
    return this.apellidos + ', ' + this.nombre
  }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort();
```

Obteniendo en la consola:

```
misAlumnos
▼ (3) [Alumno, Alumno, Alumno]
  ► 0: Alumno {nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
  ► 1: Alumno {nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
  ► 2: Alumno {nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
    length: 3
  ► [[Prototype]]: Array(0)
```

NOTA: si queremos ordenar un array de objetos por un campo numérico lo más sencillo es restar dicho campo:

```
misAlumnos.sort((alum1, alum2) => alum1.edad - alum2.edad)
```

EJERCICIO: modifica las clases Productos y Televisores para que el método que muestra los datos del producto se llame de una manera más adecuada.

EJERCICIO: Crea 5 productos y guárdalos en un array. Crea las siguientes funciones (todas reciben ese array como parámetro):

prodsSortByName: devuelve un array con los productos ordenados alfabéticamente.

prodsSortByPrice: devuelve un array con los productos ordenados por precio.

prodsTotalPrice: devuelve el importe total de los productos del array, con 2 decimales.

prodsWithLowUnits: además del array recibe como segundo parámetro un número, y devuelve un array con todos los productos de los que quedan menos de las unidades indicadas.

prodsList: devuelve una cadena que dice ‘Listado de productos:’ y en cada línea un guión y la información de un producto del array.

valueOf()

Al comparar objetos (con >, <, ...) se usa el valor devuelto por el método `.toString()`

Pero si definimos un método `.valueOf()` será este el que se usará en comparaciones:

```
class Alumno {
  ...
  valueOf() {
    return this.edad
  }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
let aat = new Alumno('Ana', 'Abad Tudela', 23)
console.log(cpo < aat)      // imprime true ya que 19<23
```

Métodos get y set en JS

Los métodos get y set no son necesarios, pero sí recomendables para modificar y/o acceder al contenido de las propiedades.

Se pueden definir para una o varias propiedades de una clase. Incluso puede definirse únicamente uno de estos métodos (set o get).

Get: lee la información (valor) de la propiedad.

Set: pone información en la propiedad.

Estos métodos no pueden llamarse igual que la propiedad, por eso pondremos propiedad (con guion bajo).

Con get y set no hay que poner paréntesis para llamar a los métodos:

Una vez definidos los métodos get y set para una propiedad, no usaremos paréntesis para invocar a esos métodos, accederemos como a sus propiedades:

- Para invocar al método get y que devuelva su valor haremos `= objeto.propiedad`

Para llamar al método set haremos: `objeto.propiedad = valor` Veamos un ejemplo para la propiedad nombre de la clase Alumno:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this._nombre = nombre;
    this._apellidos = apellidos;
    this.edad = edad;
  }

  get nombre(){
    ...
  }
}
```

```

return this._nombre;
}
set nombre(nombre){
this._nombre = nombre;
}
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
cpo.nombre = 'Carlos Luis'; //set nombre('Carlos Luis')
console.log (cpo.nombre); // get nombre() -> Carlos Luis

```

El método **set** es más necesario en el caso que queramos que se guarde la información con un formato determinado. Dentro del método **set** podemos hacer los cambios que necesitemos.

Por ejemplo, para que se guarde siempre en mayúsculas:

```

set nombre(nombre){
this._nombre = nombre.toUpperCase(); //
}

```

El método **get** es muy útil para devolver información en un formato determinado como si accediéramos a una propiedad (sin paréntesis).

Por ejemplo, para devolver en formato: Apellidos, nombre utilizaremos el método `get nombreCompleto()`

```

get nombreCompleto(){
return this._apellidos + ', ' + this.nombre;
}

console.log(cpo.nombreCompleto); // se accede sin paréntesis

```

Get y set, en conjunto, pueden ser una forma muy eficaz de obtener seguridad en cuando a validación de datos.

Agregar propiedades y métodos a los objetos de una clase (**prototype**).

En Javascript un objeto se crea a partir de otro (al que se llama *prototipo*). Así se crea una cadena de prototipos, el primero de los cuales es el objeto *null*.

Ya vimos como añadir una propiedad a un objeto. Pero si queremos que esa propiedad la tengan todos los objetos de la clase, y no queremos modificar el constructor, hay que hacerlo con **prototype**. Se le puede aplicar un valor por defecto.

Igualmente se pueden añadir métodos a la clase (también con **prototype**).

En el siguiente ejemplo se ha añadido la propiedad `email` y el método `ApellidosMayusc()` a la clase `Alumnos`.

```

class Alumno {
constructor(nombre, apellidos, edad) {
  this.nombre = nombre;
  this.apellidos = apellidos;
  this.edad = edad;
}
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);

```

```
Alumno.prototype.email='...@email.com'; // valor por defecto
console.log(cpo.email) // ...@email.com

Alumno.prototype.ApellidosMayusc= function () {
    return this.apellidos.toLowerCase();
}

console.log(cpo.ApellidosMayusc());
```

Prototipos y POO en JS5

Las versiones de Javascript anteriores a ES2015 no soportan clases ni herencia.

Este apartado está sólo para que comprendamos este código si lo vemos en algún programa, pero nosotros programaremos como hemos visto antes.

Si queremos emular en JS5 el comportamiento de las clases, para crear el constructor se crea una función con el nombre del objeto y para crear los métodos se aconseja hacerlo en el *prototipo* del objeto para que no se cree una copia del mismo por cada instancia que creemos:

```
function Alumno(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
}
Alumno.prototype.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(cpo.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Cada objeto tiene un prototipo del que hereda sus propiedades y métodos (es el equivalente a su clase, pero en realidad es un objeto que está instanciado).

Si añadimos una propiedad o método al prototipo se añade a todos los objetos creados a partir de él, lo que ahorra mucha memoria.

Uso de call y apply en Javascript

Con la palabra reservada **call** se puede llamar a un método de un objeto, desde otro objeto que no tiene ese método.

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(){
        return this.apellidos + ', ' + this.nombre;
    }
}
```

```
let azp= {
    nombre: 'Ana',
    apellidos: 'Zubiri Peláez',
}

console.log(cpo.nombreCompleto()); // Pérez Ortiz, Carlos
console.log(cpo.nombreCompleto.call(azp)); // Zubiri Peláez, Ana
```

Se pueden pasar argumentos a la llamada con **call**, en ese caso **la función llamada** tratará los argumentos como un parámetro normal.

```
console.log(cpo.nombreCompleto.call(azp, argum1, argum2));
```

Con **apply** se puede llamar a un método de un objeto con los datos de otro objeto que no tiene definido ese método. Se diferencia de call en cómo se pasan los argumentos.

En el caso de apply los argumentos se pasan en un array, no de uno en uno.

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return titulo + ' ' + this.nombre + ', ' + email;
    }
}

let azp= {
    nombre: 'Ana',
    apellidos: 'Zubiri Peláez',
}

arrayArgumentos= ['Doña', 'anzupe@email.com'];
console.log(cpo.nombreCompleto.apply(azp, arrayArgumentos));
```

La clase Object

Como todas las clases heredan de la clase Object, podemos utilizar con todos los objetos las propiedades y métodos de Object:

Ejemplo que lista los valores las propiedades de un objeto, también los métodos:

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return titulo + ' ' + this.nombre + ', ' + email;
    }
}

console.log(Object.values(cpo)); // (3) ['Carlos', 'Pérez Ortiz', f]
```

Ejemplo que lista propiedades (clave: valor) de las propiedades y métodos de un objeto. En este caso, se utiliza `Object.prototype`, que permite añadir métodos a todas las clases:

```
Object.prototype.imprime = function() {
    Object.entries(this).forEach(([key, value])=>{
        console.log(key + ': ' + value);
    })
}

let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return titulo + ' ' + this.nombre + ', ' + email;
    }
}

cpo.imprime();
```

Podemos utilizar el método `hasOwnProperty("nombre_propiedad")` para que

```
console.log(cpo.hasOwnProperty("edad")); // devuelve false porque "edad" no es una propiedad
```

Este método está heredado de **Object**. Podemos ver otros métodos del prototipo Object expandiendo la salida del objeto desde la consola:

```
console.log(cpo);
```

```
false
▼ {nombre: 'Carlos', apellidos: 'Pérez Ortiz', nombreCompleto: f} ⓘ
  apellidos: "Pérez Ortiz"
  nombre: "Carlos"
▶ nombreCompleto: f (titulo, email)
▼ [[Prototype]]: Object
  ▶ imprime: f ()
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: ...
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```



DWEC – Javascript Web Cliente.

JavaScript 04 – DOM - Document Object Model	1
Introducción.....	1
Acceso a los nodos.....	3
getElementById(id)	3
getElementsByClassName(clase).....	4
getElementsByTagName(elemento)	4
querySelector(selector).....	4
querySelectorAll(selector).....	4
Atajos	5
Acceso a nodos a partir de otros	6
Propiedades de un nodo.....	7
innerHTML	7
textContent	7
value.....	7
Manipular el árbol DOM	8
createElement.....	8
createTextNode	8
appendChild	8
insertBefore	8
removeChild.....	8
replaceChild	9
cloneNode	9
Ejemplo de creación de nuevos nodos:	9
Modificar el DOM con ChildNode	10
Atributos de los nodos	10
Estilos de los nodos	11
Atributos de clase	11

JavaScript 04 – DOM - Document Object Model

Introducción

La mayoría de las veces que se programa con Javascript es para que se ejecute en una página web mostrada por el navegador. En este contexto, y para facilitar el desarrollo de la aplicación, tenemos acceso a ciertos objetos que nos permiten interactuar con la página (DOM) y con el navegador (Browser Object Model, BOM).

El **DOM** es una estructura en árbol que representa todos los elementos HTML de la página y sus atributos.

Todo lo que contiene la página se representa como nodos del árbol y mediante el DOM podemos acceder a cada nodo, modificarlo, eliminarlo o añadir nuevos nodos de forma que cambiamos dinámicamente la página mostrada al usuario.

La raíz del árbol DOM es **document** y de este nodo cuelgan el resto de elementos HTML.

Cada elemento constituye su propio nodo y tiene subnodos con sus *atributos*, *estilos* y otros elementos HTML que contiene.

Nota: Un **elemento HTML** consiste en:

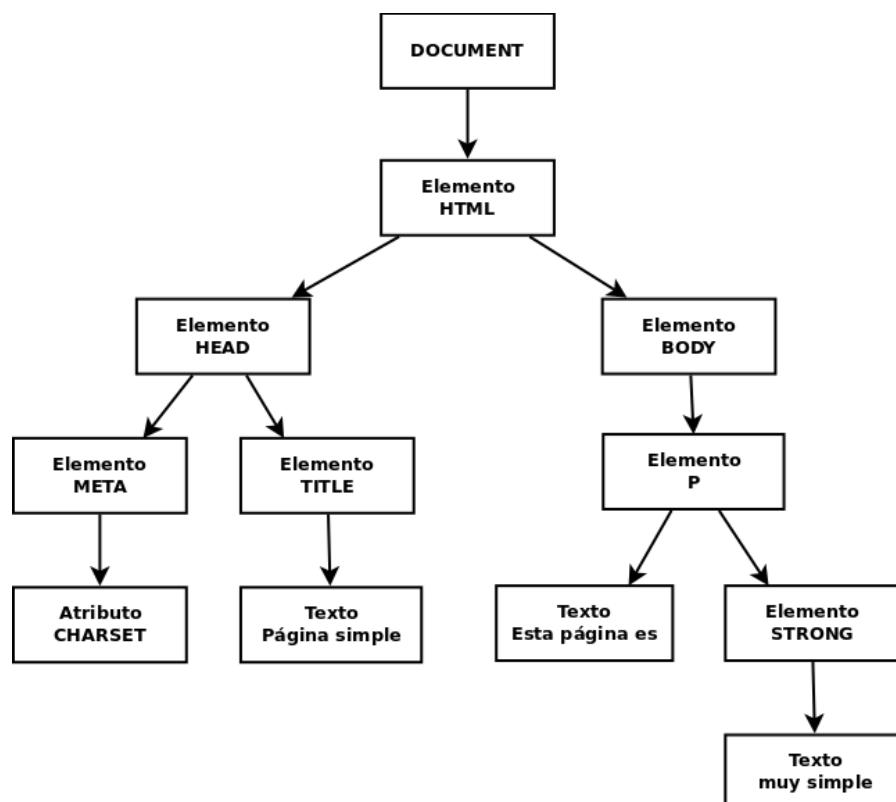
- Una **etiqueta inicial**. Opcionalmente contiene pares “atributo: valor”.
- **Contenido** del elemento (no siempre aparece).
- **Etiqueta final** o de cierre (no siempre aparece).

Es frecuente que alguien se refiera a un ‘elemento HTML’ como ‘etiqueta HTML’. Por lo que debes interpretar el significado de **etiqueta** en cada momento según el contexto.

Por ejemplo, la página HTML:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Página simple</title>
</head>
<body>
  <p>Esta página es <strong>muy simple</strong></p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



Cada elemento HTML suele originar 2 nodos:

- **Element**: correspondiente al elemento.
- **Text**: correspondiente a su contenido (lo que hay entre la etiqueta inicial y su par de cierre)

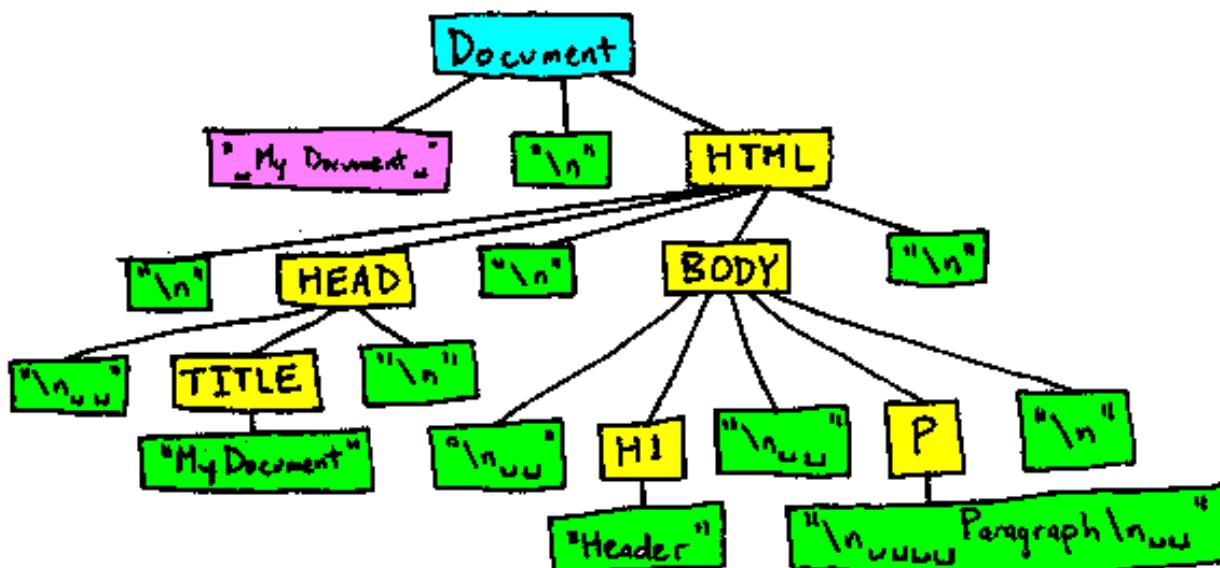
Cada nodo es un objeto con sus propiedades y métodos.

El ejemplo anterior está simplificado porque sólo aparecen los nodos de tipo **elemento**, pero en realidad también generan nodos los saltos de línea, tabuladores, espacios, comentarios, etc.

En el siguiente ejemplo podemos ver TODOS los nodos que realmente se generan. La página:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Document</title>
</head>
<body>
  <h1>Header</h1>
  <p>
    Paragraph
  </p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



Acceso a los nodos

Los principales métodos para acceder a los diferentes nodos son:

getElementById(id)

.**getElementById(id)**: devuelve el nodo con la *id* pasada. Ej.:

```
let nodo = document.getElementById('main'); // nodo contendrá el nodo cuya id es _main_
```

Cuidado de no confundir con el método `.getElementsByName(nombre)`, que devuelve los elementos con el atributo *name* especificado.

getElementsByClassName(clase)

.getElementsByClassName(clase): devuelve una colección (similar a un array) con todos los nodos de la *clase* indicada. Ej.:

```
let nodos = document.getElementsByClassName('error'); // nodos contendrá todos los nodos cuya clase es _error_
```

NOTA: las colecciones son similares a arrays (se accede a sus elementos con *[indice]*) pero no se les pueden aplicar los métodos *filter, map, ...* a menos que se conviertan a arrays con *Array.from()*

getElementsByTagName(elemento)

.getElementsByTagName(elemento): devuelve una colección con todos los nodos del tipo *elemento* HTML indicado. Ej.:

```
let nodos = document.getElementsByTagName('p'); // nodos contendrá todos los nodos de tipo _<p>_
```

querySelector(selector)

.querySelector(selector): devuelve el primer nodo seleccionado por el *selector CSS* indicado. Ej.:

```
let nodo = document.querySelector('p.error'); // nodo contendrá el primer párrafo de clase _error_
```

querySelectorAll(selector)

.querySelectorAll(selector): devuelve una *NodeList* con todos los nodos seleccionados por el *selector CSS* indicado. Ej.:

```
let nodos = document.querySelectorAll('p.error'); // nodos contendrá todos los párrafos de clase _error_
```

```
let nodo = document.querySelectorAll('#text'); // nodo contendrá el elemento con ID=text
```

Más información en: <https://developer.mozilla.org/es/docs/Web/API/Document/querySelectorAll>

NOTA: Los objetos `NodeList` son colecciones de nodos como los devueltos por propiedades como `Node.childNodes` y el método `document.querySelectorAll()` ..

NOTA: al aplicar estos métodos sobre `document` se seleccionarán sobre la página (objeto `document`). Pero podrían también aplicarse a cualquier nodo, en ese caso la búsqueda se realizaría sólo entre los descendientes de dicho nodo.

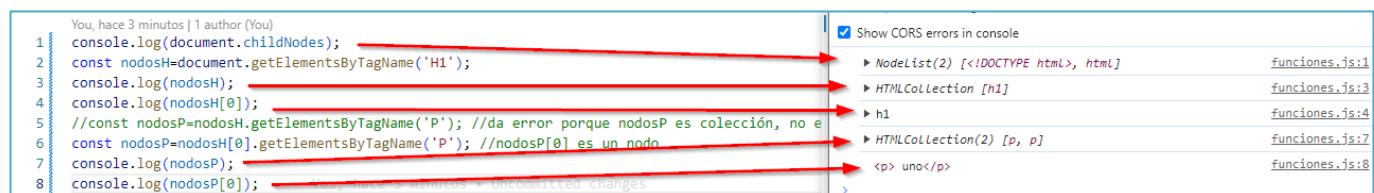
Ejemplo:

Partiendo de este .html:

```
<body>
  <h1>El DOM
    <p> uno</p>
    <p> dos</p>
  </h1>
  <script src="js/funciones.js"></script>
</body>
```

Aplicamos el siguiente código:

```
console.log(document.childNodes);
const nodosH=document.getElementsByTagName('H1');
console.log(nodosH);
console.log(nodosH[0]);
//const nodosP=nodosH.getElementsByTagName('P'); //da error porque nodosP es colección, no es nodo
const nodosP=nodosH[0].getElementsByTagName('P'); //nodosP[0] es un nodo
console.log(nodosP);
console.log(nodosP[0]);
```



Atajos

También tenemos ‘atajos’ para obtener algunos elementos comunes:

- document.documentElement: devuelve el nodo del elemento `<html>`
- document.head: devuelve el nodo del elemento `<head>`
- document.body: devuelve el nodo del elemento `<body>`
- document.title: devuelve el nodo del elemento `<title>`
- document.link: devuelve una colección con todos los hiperenlaces del documento
- document.anchor: devuelve una colección con todas las anclas del documento
- document.forms: devuelve una colección con todos los formularios del documento
- document.images: devuelve una colección con todas las imágenes del documento
- document.scripts: devuelve una colección con todos los scripts del documento

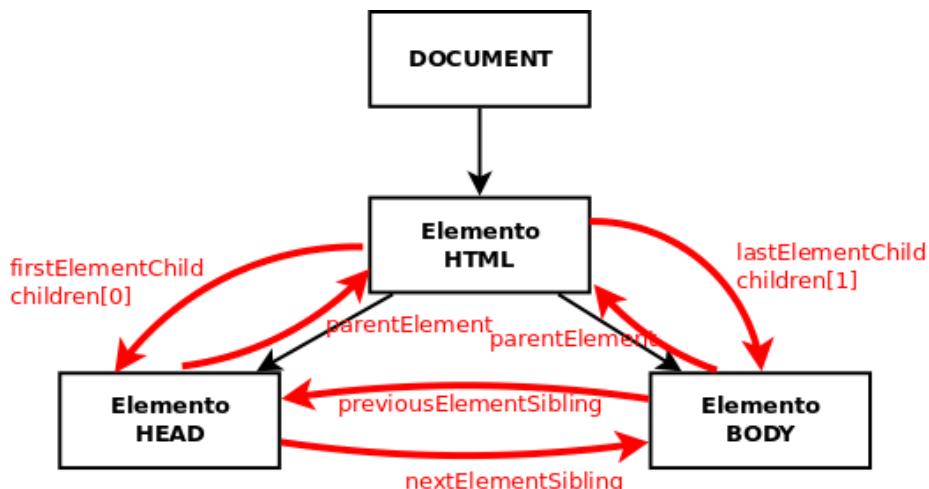
EJERCICIO: Para hacer los ejercicios de este tema descárgate [esta página de ejemplo](#) y ábrelo en tu navegador. Obtén por consola, al menos de 2 formas diferentes lo que se pide:

1. El elemento con id ‘input2’
2. La colección de párrafos
3. Lo mismo pero sólo de los párrafos que hay dentro del div ‘lipsum’
4. El formulario (ojo, no la colección con el formulario sino sólo el formulario)

5. Todos los inputs
6. Sólo los inputs con nombre 'sexo'
7. Los items de lista de la clase 'important' (sólo los LI)

Acceso a nodos a partir de otros

En muchas ocasiones queremos acceder a cierto nodo a partir de uno dado. Para ello tenemos los siguientes métodos que se aplican sobre un elemento del árbol DOM:



- `elemento.parentElement`: devuelve el elemento padre de *elemento*
- `elemento.children`: devuelve la colección con todos los elementos hijo de *elemento* (sólo elementos HTML, no comentarios ni nodos de tipo texto)
- `elemento.childNodes`: devuelve la colección con todos los hijos de *elemento*, incluyendo comentarios y nodos de tipo texto por lo que no suele utilizarse
- `elemento.firstElementChild`: devuelve el elemento HTML que es el primer hijo de *elemento*
- `elemento.firstChild`: devuelve el nodo que es el primer hijo de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.lastElementChild`, `elemento.lastChild`: igual pero con el último hijo
- `elemento.nextElementSibling`: devuelve el elemento HTML que es el siguiente hermano de *elemento*
- `elemento.nextSibling`: devuelve el nodo que es el siguiente hermano de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.previousElementSibling`, `elemento.previousSibling`: igual pero con el hermano anterior
- `elemento.hasChildNodes`: indica si *elemento* tiene o no nodos hijos
- `elemento.childElementCount`: devuelve el nº de nodos hijo de *elemento*

IMPORTANTE: a menos que interesen comentarios, saltos de página, etc ..., **siempre** hay que usar los métodos que sólo devuelven elementos HTML, no todos los nodos.

EJERCICIO: Siguiendo con la [página de ejemplo](#) obtén desde la consola, al menos de 2 formas diferentes:

1. El primer párrafo que hay dentro del div 'lipsum'
2. El segundo párrafo de 'lipsum'
3. El último item de la lista
4. El elemento *label* de 'Escoge sexo'

Propiedades de un nodo

Las principales propiedades de un nodo son:

innerHTML

`elemento.innerHTML`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, incluyendo otras etiquetas HTML. Por ejemplo, si *elemento* es el nodo:

```
<p>Esta página es <strong>muy simple</strong></p>
```

```
let contenido = elemento.innerHTML; // contenido='Esta página es <strong>muy simple</strong>'
```

textContent

`elemento.textContent`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, pero ignorando otras etiquetas HTML. Siguiendo con el ejemplo anterior:

```
let contenido = elemento.textContent; // contenido='Esta página es muy simple'
```

value

`elemento.value`: devuelve la propiedad ‘value’ de un `<input>` (en el caso de un `<input>` de tipo `text` devuelve lo que hay escrito en él).

Como los `<inputs>` no tienen etiqueta de cierre (`</input>`) no podemos usar `.innerHTML` ni `.textContent`.

Por ejemplo:

si *elem1* es el nodo `<input name="nombre">` y *elem2* es el nodo `<input type="radio" value="H">` Hombre

```
let cont1 = elem1.value; // cont1 valdría lo que haya escrito en el <input> en ese momento
let cont2 = elem2.value; // cont2="H"
```

Otras propiedades:

- `elemento.innerText`: igual que `textContent`
- `elemento.focus`: pone (sitúa) el foco en *elemento* (para inputs, etc). Para quitarle el foco `elemento.blur`
- `elemento.clientHeight / elemento.clientWidth`: devuelve el alto / ancho visible del *elemento*
- `elemento.offsetHeight / elemento.offsetWidth`: devuelve el alto / ancho total del *elemento*
- `elemento.clientLeft / elemento.clientTop`: devuelve la distancia de *elemento* al borde izquierdo / borde superior
- `elemento.offsetLeft / elemento.offsetTop`: devuelve los *píxeles* que hemos desplazado *elemento* a la izquierda / abajo

EJERCICIO: Obtén desde la consola, al menos de 2 formas:

1. El `innerHTML` de la etiqueta de ‘Escoge sexo’
2. El `textContent` de esa etiqueta
3. El valor del primer input de sexo

4. El valor del sexo que esté seleccionado (difícil, búscalo por Internet)

Manipular el árbol DOM

Vamos a ver qué métodos nos permiten cambiar el árbol DOM, y por tanto modificar la página:

createElement

`document.createElement('etiqueta')`: crea un nuevo elemento HTML con la etiqueta indicada, pero aún no se añade a la página. Ej.:

```
let nuevoLi = document.createElement('li');
```

createTextNode

`document.createTextNode('texto')`: crea un nuevo nodo de texto con el texto indicado, que luego tendremos que añadir a un nodo HTML. Ej.:

```
let textoLi = document.createTextNode('Nuevo elemento de lista');
```

appendChild

`elemento.appendChild(nuevoNodo)`: añade *nuevoNodo* como último hijo de *elemento*. Ahora ya se ha añadido a la página. Ej.:

```
nuevoLi.appendChild(textoLi);      // añade el texto creado al elemento LI creado
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
miPrimeraLista.appendChild(nuevoLi); // añade LI como último hijo de UL, es decir al
final de la Lista
```

insertBefore

`elemento.insertBefore(nuevoNodo, nodo)`: añade *nuevoNodo* como hijo de *elemento* antes del hijo *nodo*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
miPrimeraLista.insertBefore(nuevoLi, primerElementoDeLista); // añade LI al principio
de la Lista
```

removeChild

`elemento.removeChild(nodo)`: borra *nodo* de *elemento* y por tanto se elimina de la página. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
```

```
miPrimeraLista.removeChild(primerElementoDeLista); // borra el primer elemento de La
lista
// También podríamos haberlo borrado sin tener el parent con:
primerElementoDeLista.parentElement.removeChild(primerElementoDeLista);
```

replaceChild

elemento.replaceChild(nuevoNodo, viejoNodo): reemplaza *viejoNodo* con *nuevoNodo* como hijo de *elemento*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
miPrimeraLista.replaceChild(nuevoLi, primerElementoDeLista); // reemplaza el 1º
elemento de la lista con nuevoLi
```

cloneNode

elementoAClonar.cloneNode(boolean): devuelve un clon de *elementoAClonar* o de *elementoAClonar* con todos sus descendientes según le pasemos como parámetro false o true. Luego podremos insertarlo donde queramos.

OJO: Si añado con el método appendChild un nodo que estaba en otro sitio **se elimina de donde estaba** para añadirse a su nueva posición. Si quiero que esté en los 2 sitios deberá clonar el nodo y luego añadir el clon y no el nodo original.

Ejemplo de creación de nuevos nodos:

Tenemos un código HTML con un DIV que contiene 3 párrafos y vamos a añadir un nuevo párrafo al final del div con el texto ‘Párrafo añadido al final’ y otro que sea el 2º del div con el texto ‘Este es el **nuevo** segundo párrafo’:

```
<div id="articulos">
    <p>Este es el primer párrafo que tiene <strong>algo en negrita</strong>.</p>
    <p>Este era el segundo párrafo pero será desplazado hacia abajo.</p>
    <p>Y este es el último párrafo pero luego añadiremos otro después</p>
</div>
```

```
let miDiv=document.getElementById('articulos');

miDiv.innerHTML+="

Párrafo añadido al final

";

let nuevoSegundoParrafo=document.createElement('p');
nuevoSegundoParrafo.innerHTML='Este es el <strong>nuevo</strong> segundo párrafo';

let segundoParrafo=miDiv.children[1];
miDiv.insertBefore(nuevoSegundoParrafo, segundoParrafo);
```

Resultado:

```
Este es el primer párrafo que tiene algo en negrita.  
Este es el nuevo segundo párrafo  
Este era el segundo párrafo pero será desplazado hacia abajo.  
Y este es el último párrafo pero luego añadiremos otro después  
Párrafo añadido al final
```

Si utilizamos la propiedad **innerHTML** el código a usar es mucho más simple:

```
let ultimoParrafo = document.createElement('p');  
ultimoParrafo.innerHTML = 'Párrafo añadido al final';  
miDiv.appendChild(ultimoParrafo);
```

OJO: La forma de añadir el último párrafo (línea #3: `miDiv.innerHTML+= '<p>Párrafo añadido al final</p>';`) aunque es válida no es muy eficiente ya que obliga al navegador a volver a pintar TODO el contenido de miDIV. La forma correcta de hacerlo sería:

```
let ultimoParrafo = document.createElement('p');  
ultimoParrafo.innerHTML = 'Párrafo añadido al final';  
miDiv.appendChild(ultimoParrafo);
```

Así sólo debe repintar el párrafo añadido, conservando todo lo demás que tenga *miDiv*.

Podemos ver más ejemplos de creación y eliminación de nodos en [W3Schools](#).

EJERCICIO: Añade a la página:

1. Un nuevo párrafo al final del DIV '*lipsum*' con el texto “Nuevo párrafo **añadido** por javascript” (fíjate que una palabra está en negrita)
2. Un nuevo elemento al formulario tras el '*Dato 1*' con la etiqueta '*Dato 1 bis*' y el INPUT con id '*input1bis*' que al cargar la página tendrá escrito “Hola”

Modificar el DOM con ChildNode

Childnode es una interfaz que permite manipular del DOM de forma más sencilla pero no está soportada en los navegadores Safari de IOS. Incluye los métodos:

- `elemento.before(nuevoNodo)`: añade el *nuevoNodo* pasado antes del nodo *elemento*
- `elemento.after(nuevoNodo)`: añade el *nuevoNodo* pasado después del nodo *elemento*
- `elemento.replaceWith(nuevoNodo)`: reemplaza el nodo *elemento* con el *nuevoNodo* pasado
- `elemento.remove()`: elimina el nodo *elemento*

Atributos de los nodos

Podemos ver y modificar los valores de los atributos de cada elemento HTML y también añadir o eliminar atributos:

- `elemento.attributes`: devuelve un array con todos los atributos de *elemento*

- `elemento.hasAttribute('nombreAtributo')`: indica si *elemento* tiene o no definido el atributo *nombreAtributo*
- `elemento.getAttribute('nombreAtributo')`: devuelve el valor del atributo *nombreAtributo* de *elemento*. Para muchos elementos este valor puede directamente con `elemento.getAttribute`.
- `elemento.setAttribute('nombreAtributo', 'valor')`: establece *valor* como nuevo valor del atributo *nombreAtributo* de *elemento*. También puede cambiarse el valor directamente con `elemento.setAttribute=valor`.
- `elemento.removeAttribute('nombreAtributo')`: elimina el atributo *nombreAtributo* de *elemento*

A algunos atributos comunes como `id`, `title` o `className` (para el atributo `class`) se puede acceder y cambiar como si fueran una propiedad del elemento (`elemento.getAttribute`). Ejemplos:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
miPrimeraLista.id = 'primera-lista';
// es equivalente ha hacer:
miPrimeraLista.setAttribute('id', 'primera-lista');
```

Estilos de los nodos

Los estilos están accesibles como el atributo `style`. Cualquier estilo es una propiedad de dicho atributo, pero con la sintaxis *camelCase* en vez de *kebab-case*.

Por ejemplo, para cambiar el color de fondo (propiedad `background-color`) y ponerle el color *rojo* al elemento *miPrimeraLista* haremos:

```
miPrimeraLista.style.backgroundColor = 'red';
```

De todas formas, normalmente **NO CAMBIAREMOS ESTILOS** a los elementos, sino que les pondremos o quitaremos clases que harán que se le apliquen o no los estilos definidos para ellas en el CSS.

Atributos de clase

Ya sabemos que el aspecto de la página debe configurarse en el CSS por lo que no debemos aplicar atributos `style` al HTML. En lugar de ello les ponemos clases a los elementos que harán que se les aplique el estilo definido para dicha clase.

Como es algo muy común en lugar de utilizar las instrucciones de `elemento.setAttribute('className', 'destacado')` o directamente `elemento.className='destacado'` podemos usar la propiedad `classList` que devuelve la colección de todas las clases que tiene el elemento.

Por ejemplo, si *elemento* es `<p class="destacado direccion">....</p>`:

```
let clases=elemento.classList; // clases=['destacado', 'direccion'], OJO es una colección, no un Array
```

Además, dispone de los métodos:

add

.add(clase): añade al elemento la clase pasada (si ya la tiene no hace nada). Ej.:

```
elemento.classList.add('primero'); // ahora elemento será <p class="destacado direccion primero">...
```

remove

.remove(clase): elimina del elemento la clase pasada (si no la tiene no hace nada). Ej.:

```
elemento.classList.remove('direccion'); // ahora elemento será <p class="destacado primero">...
```

toogle

.toogle(clase): añade la clase pasada si no la tiene o la elimina si la tiene ya. Ej.:

```
elemento.classList.toggle('destacado'); // ahora elemento será <p class="primero">...
elemento.classList.toggle('direccion'); // ahora elemento será <p class="primero direccion">...
```

contains

.contains(clase): dice si el elemento tiene o no la clase pasada. Ej.:

```
elemento.classList.contains('direccion'); // devuelve true
```

replace

.replace(oldClase, newClase): reemplaza del elemento una clase existente por una nueva. Ej.:

```
elemento.classList.replace('primero', 'ultimo'); // ahora elemento será <p class="ultimo direccion">...
```

Ten en cuenta que NO todos los navegadores soportan *classList* por lo que si queremos añadir o quitar clases en navegadores que no lo soportan debemos hacerlo con los métodos estándar, por ejemplo para añadir la clase 'rojo':

```
let clases = elemento.className.split(" ");
if (clases.indexOf('rojo') == -1) {
  elemento.className += ' ' + 'rojo';
}
```

DWEC – Javascript Web Cliente.

JavaScript 05 – Browser Object Model (BOM).....	1
Introducción.....	1
Timers	1
Objetos del BOM.....	3
Objeto window.....	3
Diálogos.....	4
Objeto location	4
Objeto history	5
Otros objetos	5

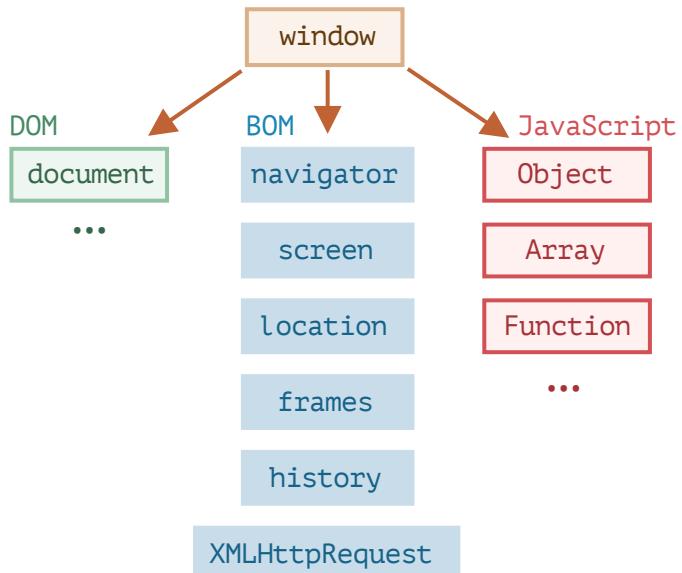
JavaScript 05 – Browser Object Model (BOM)

Introducción

Hemos visto cómo interactuar con la página (DOM), ahora veremos cómo acceder a objetos que nos permitan interactuar con el navegador utilizando el BOM (*Browser Object Model*).

Usando los objetos BOM podemos:

- Abrir, cambiar y cerrar ventanas.
- Ejecutar código transcurrido cierto tiempo (*timers*).
- Obtener información del navegador.
- Ver y modificar propiedades de la pantalla.
- Gestionar cookies, ...



Timers

Permiten ejecutar código en el futuro (cuando transcurran los milisegundos indicados). Hay 2 tipos:

- `setTimeout(función, milisegundos)`: ejecuta la función especificada **una sola vez**, cuando transcurran los milisegundos indicados.
- `setInterval(función, milisegundos)`: ejecuta la función especificada **cada vez que transcurran** los milisegundos indicados, hasta que sea cancelado el *timer*.

Se pueden pasar más parámetros a las funciones, se ponen tras los milisegundos, y serán los parámetros que recibirá la función a ejecutar.

Ambas funciones devuelven un identificador que nos permitirá cancelar la ejecución del código, con:

- clearTimeout(identificador)
- clearInterval(identificador)

Ejemplo:

```
const idTimeout = setTimeout(() => console.log('Timeout que se ejecuta al cabo de 1 seg.'), 1000);

let i = 1;
const idInterval = setInterval(() => {
    console.log('Interval cada 3 seg. Ejecución nº: ' + i++);
    if (i === 5) {
        clearInterval(idInterval);
        console.log('Fin de la ejecución del Interval');
    }
}, 3000);
```

EJERCICIO: Prueba a ejecutar cada una de esas funciones.

En lugar de definir la función a ejecutar podemos llamar a una función que ya exista:

```
function showMessage() {
    console.log('Timeout que se ejecuta al cabo de 1 seg.')
}

const idTimeout=setTimeout(showMessage, 1000);
```

Pero en ese caso hay que poner sólo el nombre de la función, sin (), ya que si los ponemos se ejecutaría la función en ese momento y no transcurrido el tiempo indicado.

Si necesitamos pasar algún parámetro a la función, los añadiremos como parámetros de setTimeout o setInterval después del intervalo.

Ejemplo:

```
function showMessage(msg) {
    alert(msg)
}

const idTimeout = setTimeout(showMessage, 1000, 'Timeout que se ejecuta al cabo de 1 seg.');
```

Otro ejemplo:

```
function myCallback(a, b) {
    // Tu código debe ir aquí
    // Los parámetros son totalmente opcionales
    console.log(a);
    console.log(b);
}

const intervalID = setInterval(myCallback, 500, 'parámetro 1', 'parámetro 2');
```

Objetos del BOM

Al contrario que para el DOM, no existe un estándar de BOM pero sus objetos (window, location, history, etc) son bastante parecidos en los diferentes navegadores.

Objeto window

Representa la ventana del navegador y es el objeto principal. De hecho puede omitirse al llamar a sus propiedades y métodos, por ejemplo, el método `setTimeout()` es en realidad `window.setTimeout()`.

Sus principales propiedades y métodos son:

- `.name`: nombre de la ventana actual
- `.statusbar`: valor de la barra de estado
- `.screenX/.screenY`: distancia de la ventana a la esquina izquierda/superior de la pantalla
- `.outerWidth/.outerHeight`: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- `.innerWidth/.innerHeight`: ancho/alto útil del documento, sin la toolbar y la scrollbar
- `.open(url, nombre, opciones)`: abre una nueva ventana. Devuelve el nuevo objeto ventana.

Las principales **opciones de `.open()`** (lista de ítems separados por comas sin espacios) son:

- `.toolbar`: si tendrá barra de herramientas
- `.location`: si tendrá barra de dirección
- `.directories`: si tendrá botones Adelante/Atrás
- `.status`: si tendrá barra de estado
- `.menubar`: si tendrá barra de menú
- `.scrollbar`: si tendrá barras de desplazamiento
- `.resizable`: si se puede cambiar su tamaño
- `.width=px/.height=px`: ancho/alto
- `.left=px/.top=px`: posición izq/sup de la ventana

Más información en https://www.w3schools.com/jsref/met_win_open.asp

- `.opener`: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con `open`)
- `.close()`: la cierra (pide confirmación, a menos que la hayamos abierto con `open`)
- `.moveTo(x,y)`: la mueve a las coord indicadas
- `.moveBy(x,y)`: la desplaza los px indicados
- `.resizeTo(x,y)`: especifica el ancho y alto indicados
- `.resizeBy(x,y)`: añade esos pixeles de ancho/alto
- `.pageXOffset / pageYOffset`: scroll actual de la ventana horizontal / vertical
- Otros métodos: `.back()`, `.forward()`, `.home()`, `.stop()`, `.focus()`, `.blur()`, `.find()`, `.print()`, ...

Más información en https://www.w3schools.com/jsref/obj_window.asp

NOTA: por seguridad no se puede mover una ventana fuera de la pantalla, ni darle un tamaño menor de 100x100 px. Tampoco se puede mover una ventana no abierta con `.open()`, o si tiene varias pestañas.

EJEMPLO:

- a) Abrir una nueva ventana de dimensiones 500x200px en la posición (200,100)
- b) Escribir en la nueva ventana (con `document.write`) un título H1 que diga ‘Hola javascritos del Claudio’.
- c) Al hacer clic sobre un botón de la ventana inicial, que la ventana se desplace 40 px a la izquierda y 50 hacia abajo
- d) Al hacer clic sobre otro botón de la ventana incicial, que se cierre la nueva ventana.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="">
    <input type="button" value="Deslazar" onclick="desplazar()">
    <input type="button" value="Cerrar" onclick="cerrar()">
  </form>
  <script src="js/bom1.js"></script>
</body>
</html>
```

```
function desplazar(){
  nuevaVentana.moveBy(40,50);
}

const cerrar = ()=> {
  nuevaVentana.close();
}

const nuevaVentana=window.open("", "_blank", "width= 500,height=200,left=200,top=100");
nuevaVentana.document.write("<h1>Hola javascritos del Claudio</h1>");
```

EJERCICIO: Haz que a los 2 segundos de abrir la página se abra un *popup* con un mensaje de bienvenida. Esta ventana tendrá en su interior un botón Cerrar que permitirá que el usuario la cierre haciendo clic en él. Tendrá el tamaño justo para visualizar el mensaje y no tendrá barras de scroll, ni de herramientas, ni de dirección... únicamente el mensaje.

Diálogos

Hay 3 métodos del objeto *window* que ya conocemos y que nos permiten abrir ventanas de diálogo con el usuario:

- *window.alert(mensaje)*: muestra un diálogo con el mensaje indicado y un botón de ‘Aceptar’
- *window.confirm(mensaje)*: muestra un diálogo con el mensaje indicado y botones de ‘Aceptar’ y ‘Cancelar’. Devuelve *true* si se ha pulsado el botón de aceptar del diálogo y *false* si no.
- *window.prompt(mensaje [, valor predeterminado])*: muestra un diálogo con el mensaje indicado, un cuadro de texto (vacío o con el valor predeterminado indicado) y botones de ‘Aceptar’ y ‘Cancelar’. Si se pulsa ‘Aceptar’ devolverá un *string* con el valor que haya en el cuadro de texto y si se pulsa ‘Cancelar’ o se cierra devolverá *null*.

Objeto location

Contiene información sobre la URL actual del navegador y podemos modificarla. Sus principales propiedades y métodos son:

- `.href`: devuelve la URL actual completa
- `.protocol`, `.host`, `.port`: devuelve el protocolo, host y puerto respectivamente de la URL actual
- `.pathname`: devuelve la ruta al recurso actual
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

EJERCICIO:

- muestra la ruta completa de la página actual
- muestra el servidor de esta página
- carga la página de Google usando el objeto `location`

```
<script>
    console.log( window.location.href);
    console.log(location.host);
    console.log(location.assign('https://www.google.es/'));
</script>
```

Objeto history

Permite acceder al historial de páginas visitadas y navegar por él:

- `.length`: muestra el número de páginas almacenadas en el historial
- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página
- `.go(num)`: se mueve *num* páginas hacia adelante en el historial (si *num* es positivo), o hacia atrás (si *num* es negativo).

EJERCICIO: Vuelve a la página anterior**Otros objetos**

Los otros objetos que incluye BOM son:

- [`document`](#): el objeto `document` que hemos visto en el DOM
- [`navigator`](#): nos informa sobre el navegador y el sistema en que se ejecuta
 - `.userAgent`: muestra información sobre el navegador que usamos
 - `.platform`: muestra información sobre la plataforma sobre la que se ejecuta
 - ...
- [`screen`](#): nos da información sobre la pantalla
 - `.width/.height`: ancho/alto total de la pantalla (resolución)
 - `.availWidth/.availHeight`: igual pero excluyendo la barra del S.O.
 - ...

EJERCICIO: obtén todas las propiedades `width/height` y `availWidth/availHeight` del objeto `screen`. Compáralas con las propiedades `innerWidth/innerHeight` y `outerWidth/outerHeight` de `window`.



DWEC – Javascript Web Cliente.

JavaScript 06 – Eventos.....	1
Introducción.....	1
Escuchar un evento utilizando un escuchador o <i>listener</i>	1
Event listeners.....	2
Tipos de eventos	3
Eventos de página.....	3
Eventos de ratón.....	3
Eventos de teclado.....	4
Eventos de toque	4
Eventos de formulario.....	4
Los objetos event y this	4
event	4
this	6
Bindeo del objeto this	7
Propagación de eventos (bubbling)	8
innerHTML y escuchadores de eventos	10
Eventos personalizados	11

JavaScript 06 – Eventos

Introducción

Los eventos permiten detectar acciones que realiza el usuario, o cambios que suceden en la página, y reaccionar como respuesta.

Existen muchos eventos diferentes, se puede ver la lista en [W3schools](#). Nos centraremos en los más comunes.

Javascript nos permite ejecutar código cuando se produce un evento, por ejemplo, el evento *click* del ratón, asociando al mismo una función. Hay varias formas de hacerlo.

Escuchar un evento utilizando un escuchador o *listener*.

La primera manera “estándar” de asociar código a un evento era añadiendo un atributo con el nombre del evento a escuchar (con ‘on’ delante) en el elemento HTML.

Por ejemplo, para ejecutar código al producirse el evento ‘click’ sobre un botón se escribía:

```
<input type="button" id="boton1" onclick="alert('Se ha pulsado');" />
```

Una mejora era llamar a una función que contenía el código:

```
<input type="button" id="boton1" onclick="clicked()" />
```

```
function clicked() {
    alert('Se ha pulsado');
}
```

Esto “ensuciaba” con código la página HTML, por lo que se creó el modelo de registro de eventos tradicional que permitía asociar a un elemento HTML una propiedad con el nombre del evento a escuchar (con ‘on’ delante). En el caso anterior sería:

```
document.getElementById('boton1').onclick = function () {
    alert('Se ha pulsado');
}
...
```

NOTA: hay que tener cuidado porque si se ejecuta el código antes de que se haya creado el botón estaremos asociando la función al evento *click* de un elemento que aún no existe, así que no hará nada.

Para evitarlo, es conveniente poner el código que atiende a los eventos dentro de una función que se ejecute al producirse el evento *load* de la ventana. Este evento se produce cuando se han cargado todos los elementos HTML de la página y se ha creado el árbol DOM. (*También podemos evitarlo si el script es llamado al final del BODY*).

Lo mismo habría que hacer con cualquier código que modifique el árbol DOM. El código correcto sería:

```
window.onload = function() {
    document.getElementById('boton1').onclick = function() {
        alert('Se ha pulsado');
    }
}
```

Event listeners

La forma recomendada de escuchar un evento es usando el modelo avanzado de registro de eventos del W3C.

Se usa el método `addEventListener` que recibe:

- como primer parámetro el nombre del evento a escuchar (sin ‘on’)
- y como segundo parámetro la función a ejecutar (OJO, sin paréntesis) cuando se produzca el evento:

```
document.getElementById('boton1').addEventListener('click', pulsado);
...
function pulsado() {
    alert('Se ha pulsado');
})
```

Habitualmente se usan funciones anónimas, ya que no necesitan ser llamadas desde fuera del escuchador:

```
document.getElementById('boton1').addEventListener('click', function() {
    alert('Se ha pulsado');
});
```

Si queremos pasar algún parámetro a la función escuchadora, que a veces es conveniente, debemos usar funciones anónimas como escuchadores de eventos:

```
<button id="acepto">Aceptar</button>
```

```
window.addEventListener('load', function() {
    document.getElementById('acepto').addEventListener('click', function() {
        alert('Se ha aceptado');
    })
})
```

NOTA: igual que antes, debemos estar seguros de que se ha creado el árbol DOM antes de poner un escuchador, por lo que se recomienda ponerlos siempre dentro de la función asociada al evento `window.onload`. O mejor: `window.addEventListener('load', ...)` como se ve en el ejemplo anterior.

Una ventaja de este método es que podemos poner varios escuchadores para el mismo evento y se ejecutarán todos ellos.

Para eliminar un escuchador se usa el método `removeEventListener`.

```
document.getElementById('acepto').removeEventListener('click', aceptado);
```

NOTA: no se puede quitar un escuchador si hemos usado una función anónima. Para quitarlo debemos usar como escuchador una función con nombre.

Tipos de eventos

Según qué o dónde se produzca un evento, estos se clasifican en:

Eventos de página

Se producen en el documento HTML, normalmente en el BODY:

- **load**: se produce cuando termina de cargarse la página (cuando ya está construido el árbol DOM). Es útil para hacer acciones que requieran que el DOM esté cargado como modificar la página o poner escuchadores de eventos
- **unload**: al destruirse el documento (ej. cerrar)
- **beforeUnload**: antes de destruirse (podríamos mostrar un mensaje de confirmación)
- **resize**: si cambia el tamaño del documento (porque se redimensiona la ventana)

Eventos de ratón

Los produce el usuario con el ratón:

- **click / dblclick**: cuando se hace *click/doble click* sobre un elemento
- **mousedown / mouseup**: al *pulsar/soltar* cualquier botón del ratón
- **mouseenter / mouseleave**: cuando el puntero del ratón *entra/sale* del elemento (tb. podemos usar *mouseover/mouseout*)
- **mousemove**: se produce continuamente mientras el puntero se *movea* dentro del elemento

NOTA: si hacemos doble click sobre un elemento, la secuencia de eventos que se produciría es: *mousedown -> mouseup -> click -> mousedown -> mouseup -> click -> dblclick*

EJERCICIO:

- a) Pon un escuchador al botón 1 de la [página de ejemplo de DOM](#) para que al hacer click se muestre el un alert con ‘Click sobre botón 1’
- b) Pon otro escuchador al mismo botón para que se abra otra ventana nueva (de 200 px de ancho y 100 de alto) con un texto dentro que reze “Nueva ventana emergente”. **Nota:** Comprueba si hay diferencias si se abre la página desde “Live Server” o directamente como archivo local.
- c) Pon otro *listener* al mismo botón para que al pasar el ratón sobre él se muestre debajo de los botones un párrafo en rojo con la frase “Se va a abrir una ventana nueva”.
- d) Pon otro escuchador al mismo botón que al salir el cursor del ratón, desaparezca el párrafo del apartado anterior.
- e) Pon un escuchador al botón 2 que desactive el escuchador del primer apartado.

Eventos de teclado

Los produce el usuario al usar el teclado:

- **keydown:** se produce al presionar una tecla y se repite continuamente si la tecla se mantiene pulsada
- **keyup:** cuando se deja de presionar la tecla
- **keypress:** acción de pulsar y soltar (sólo se produce en las teclas alfanuméricas)

NOTA: el orden de secuencia de los eventos es: *keyDown -> keyPress -> keyUp*

Eventos de toque

Se producen al usar una pantalla táctil:

- **touchstart:** se produce cuando se detecta un toque en la pantalla táctil
- **touchend:** cuando se deja de pulsar la pantalla táctil
- **touchmove:** cuando un dedo es desplazado a través de la pantalla
- **touchcancel:** cuando se interrumpe un evento táctil.

Eventos de formulario

Se producen en los formularios:

- **focus / blur:** al obtener/perder el foco el elemento.
- **change:** al perder el foco un *<input>* o *<textarea>* si ha cambiado su contenido, o al cambiar de valor un *<select>* o un *<checkbox>*.
- **input:** al cambiar el valor de un *<input>* o *<textarea>*, Se produce cada vez que escribimos una letra en estos elementos.
- **select:** al cambiar el valor de un *<select>* o al seleccionar texto de un *<input>* o *<textarea>*.
- **submit / reset:** al enviar/recargar un formulario.

Los objetos event y this

Al producirse un evento, se generan automáticamente en su función manejadora 2 objetos: **this** y **event**.

event

event: es un objeto, y la función escuchadora lo recibe como parámetro. Tiene propiedades y métodos que nos dan información sobre el evento, como:

- **.type:** qué evento se ha producido (click, submit, keyDown, ...)
- **.target:** el elemento donde se produjo el evento. Puede ser *this* o un descendiente de *this*, como se ve en el ejemplo que hay un poco más abajo.
- **.currentTarget:** Identifica el target (objetivo) actual del evento, ya que el evento atraviesa el DOM. Siempre hace referencia al elemento al cual el controlador del evento fue asociado, a diferencia de *event.target*, que identifica el elemento en el que se produjo el evento.

Ejemplo: Tenemos un elemento *P* al que le ponemos un escuchador de ‘click’ que dentro tiene un elemento *STRONG*.

Si hacemos *_click* sobre el elemento *STRONG*: **event.target** valdrá el *STRONG* que es donde hemos hecho click (está dentro de *<p>*), pero tanto para *P* como para *STRONG* **.event.currentTarget** valdrá el elemento *<p>* (que es quien tiene el escuchador que se está ejecutando).

- **.relatedTarget:** en un evento ‘mouseover’: **event.target** es el elemento donde ha entrado el puntero del ratón y **event.relatedTarget** el elemento del que ha salido. En un evento ‘mouseout’: sería al revés.
- **.cancelable:** si el evento puede cancelarse. En caso afirmativo se puede llamar a **event.preventDefault()** para cancelarlo
- **.preventDefault():** si un evento tiene un escuchador asociado se ejecuta el código de dicho escuchador y después el navegador realiza la acción que correspondería por defecto al evento si no tuviera escuchador.

Por ejemplo: un escuchador del evento *click* sobre un hiperenlace hará que se ejecute su código y después saltará a la página indicada en el *href* del hiperenlace. Este método cancela la acción por defecto del navegador para el evento.

Otro ejemplo de uso de este método: si el evento era el *submit* de un formulario éste no se enviará, o si era un *click* sobre un hiperenlace no se irá a la página indicada en él.

- **.stopPropagation():** Como por defecto un evento se produce sobre un elemento y todos sus padres, al usar este método se evita esta propagación.

Por ejemplo: si hacemos click en un ** que está en un *<p>* que está en un *<div>* que está en el *BODY*, el evento se va propagando por todos estos elementos y saltarían los escuchadores asociados a todos ellos (si los hubiera). Si algún escuchador llama a este método, el evento no se propagará a los demás elementos padre.

Un evento, dependiendo del tipo, puede tener más propiedades:

eventos de ratón:

- **.button:** qué botón del ratón se ha pulsado (0: izq, 1: rueda; 2: dcho).
- **.screenX / .screenY:** las coordenadas del ratón respecto a la pantalla.
- **.clientX / .clientY:** las coordenadas del ratón respecto a la ventana cuando se produjo el evento.
- **.pageX / .pageY:** las coordenadas del ratón respecto al documento (si se ha hecho un scroll será el clientX/Y más el scroll).
- **.offsetX / .offsetY:** las coordenadas del ratón respecto al elemento sobre el que se produce el evento.
- **.detail:** si se ha hecho click, doble click o triple click

eventos de teclado:

Son los más incompatibles entre diferentes navegadores. En el teclado hay teclas normales y especiales (Alt, Ctrl, Shift, Enter, Tab, flechas, Supr, ...). En la información del teclado hay que distinguir entre el código del carácter pulsado (e=101, E=69, €=8364) y el código de la tecla pulsada (para los 3 caracteres es el 69 ya que se pulsa la misma tecla). Las principales propiedades de *event* son:

- **.key**: devuelve el nombre de la tecla pulsada
- **.which**: devuelve el código de la tecla pulsada
- **.keyCode / .charCode**: código de la tecla pulsada y del carácter pulsado (según navegadores)
- **.shiftKey / .ctrlKey / .altKey / .metaKey**: si está o no pulsada la tecla SHIFT / CTRL / ALT / META. Esta propiedad también la tienen los eventos de ratón

NOTA: a la hora de saber qué tecla ha pulsado el usuario es conveniente tener en cuenta:

- Para saber qué carácter se ha pulsado lo mejor es usar la propiedad *key* o *charCode* de *keyPress*, pero varía entre navegadores.
- Para saber la tecla especial pulsada, mejor usar el *key* o el *keyCode* de *keyUp*.
- Hay que capturar sólo lo que sea necesario, se producen muchos eventos de teclado.
- Para obtener el carácter a partir del código, se aconseja utilizar: `String.fromCharCode(n1, n2, ...,)`
- Lo mejor para familiarizarse con los diferentes eventos es consultar los [ejemplos de w3schools](#).

EJERCICIO A: Pon un escuchador al BODY de la [página de ejemplo](#) para que al mover el ratón en cualquier punto de la ventana del navegador, se muestre en los distintos DIV la posición del puntero respecto del navegador y respecto de la página.

EJERCICIO B: Pon un listener al BODY de la [página de ejemplo](#) para que al pulsar cualquier tecla nos muestre en un párrafo el *key* y el *keyCode* de la tecla pulsada. Pruébalo con diferentes teclas.

this

this: siempre hace referencia al elemento que contiene el código en donde se encuentra la variable *this*. En el caso de una función escuchadora será el elemento que tiene el escuchador que ha recibido el evento.

Ojo!: No usar funciones flecha. Más información en el documento JavaScript - Anexo - “Uso de this en contexto”.

Veamos un ejemplo:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Ejemplo 1 DOM</title>
  </head>
  <body>
    <h1 class="important">Ejemplo de eventos con this</h1>

    <form action="#">
      <label for="input1">
```

```

    >Dato 1 <input id="input1" type="text" size="20" /><br />
</label>
<label for="input2">
    >Dato 2 <input id="input2" type="text" size="20" /><br />
</label>
<label for="input3">Dato 3
    <input id="input3" type="text" size="20" /><br />
</label>
</form>
<script src="js/this.js"></script>
</body>
</html>

```

```

const arrayInputs=Array.from(document.getElementsByTagName('input'));

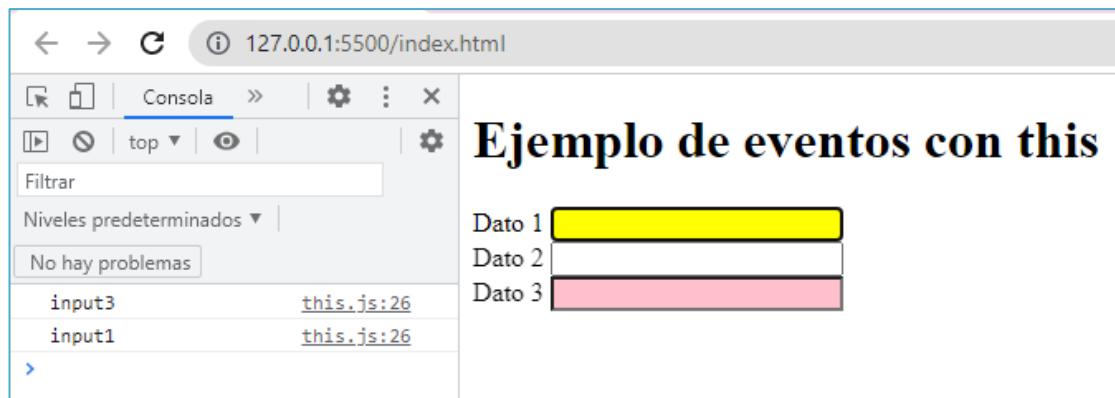
function pintar(even){
    console.log(even.target.id);
    this.style.backgroundColor='yellow';
}

function pintarRosa(even){
    this.style.backgroundColor='pink';
}

arrayInputs.forEach(element => {
    element.addEventListener('focus', pintar); //Al hacer clic los pone en amarillo
    element.addEventListener('blur', pintarRosa); // Al salir los deja rosa (para siempre)
});

```

Ejemplo de ejecución:



Bindeo del objeto this

En ocasiones no queremos que *this* sea el elemento sobre quien se produce el evento, sino que queremos conservar el valor que tenía *this* antes de entrar a la función escuchadora.

Por ejemplo: Si la función escuchadora es un método de una clase, en *this* tenemos el objeto de la clase sobre el que estamos actuando, pero al entrar en la función perdemos esa referencia.

El método `.bind()` nos permite pasarle a una función el valor que queremos darle a la variable `this` dentro de dicha función.

Por defecto a una función escuchadora de eventos se le *bindea* el valor de `event.currentTarget`. Si queremos que tenga otro valor se lo indicamos con `.bind()`:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(variable));
```

En este ejemplo, el valor de `this` dentro de la función `aceptado` será `variable`.

En el ejemplo que habíamos comentado de un escuchador dentro de una clase, para mantener el valor de `this` y que haga referencia al objeto sobre el que estamos actuando haríamos:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(this));
```

por lo que el valor de `this` dentro de la función `aceptado` será el mismo que tenía fuera, es decir, el objeto.

Podemos *bindear*, es decir, pasarle a la función escuchadora más variables, declarándolas como parámetros de `bind`. El primer parámetro será el valor de `this`, y los demás serán parámetros que recibirá la función antes de recibir el parámetro `event`, que será el último. Por ejemplo:

```
document.getElementById('acepto').removeEventListener('click', aceptado.bind(var1, var2, var3));
...
function aceptado(param1, param2, event) {
    // Aquí dentro tendremos los valores
    // this = var1
    // param1 = var2
    // param2 = var3
    // event es el objeto con la información del evento producido
}
```

Ejercicio: tarea 0xx : Una tabla de varia filas y columnas que vaya metiendo los valores de un array según se haga clic en la tabla.

Propagación de eventos (bubbling)

Normalmente en una página web los elementos HTML se solapan unos con otros, por ejemplo: un `` está en un `<p>` que está en un `<div>` que está en el `<body>`. Si ponemos un escuchador del evento `click` a todos ellos se ejecutarán todos ellos, pero ¿en qué orden?

Pues el W3C estableció un modelo en el que primero se disparan los eventos de fuera hacia dentro (primero el `<body>`) y al llegar al más interno (el ``) se vuelven a disparar de nuevo, pero de dentro hacia afuera.

La primera fase se conoce como **fase de captura** y la segunda como **fase de burbujeo**.

Cuando ponemos un escuchador con `addEventListener` el tercer parámetro indica en qué fase debe dispararse:

- **true**: en fase de captura
- **false** (valor por defecto): en fase de burbujeo

Ejemplo:

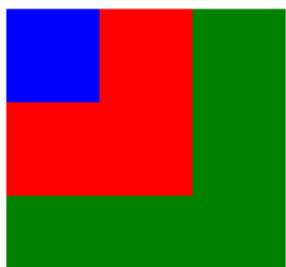
```
<div id="divVerde" style="background-color: green; width: 150px; height: 150px;">
```

```
<div id="divRojo" style="background-color: red; width: 100px; height: 100px;">
  <div id="divAzul" style="background-color: blue; width: 50px; height: 50px;"></div>
</div>
<p id="info"></p>
```

```
let divClick = function(event) {
  // eventPhase: 1 -> capture, 2 -> target (objetivo), 3 -> bubble
  document.getElementById('info').innerHTML+="Has pulsado: " + this.id + ". Fase: " +
  event.eventPhase + ", this: " + this.id + ", event.target: " + event.target.id + ", "
  event.currentTarget: " + event.currentTarget.id + "<br>";
};

let divVerde = document.getElementById("divverde");
let divRojo = document.getElementById("divRojo");
let divAzul = document.getElementById("divAzul");
divVerde.addEventListener('click', divClick);
divRojo.addEventListener('click', divClick);
divAzul.addEventListener('click', divClick);
```

Veremos algo similar a :



CASO A - Tercer parámetro por defecto (false)

Haciendo clic en el div Azul: veremos un resultado así:

```
Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
Has pulsado: divRojo. Fase: 3, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divVerde. Fase: 3, this: divVerde, event.target: divAzul, event.currentTarget: divVerde
```

Primero responde Azul en fase 2 (objetivo) y luego responden Rojo y Verde en fase 3 (burbujeo).

Observa que **event.target** siempre es Azul, mientras que **event.currentTarget** va cambiando.

CASO B - Tercer parámetro por defecto (false)

Sin embargo, si al método `.addEventListener` le pasamos un tercer parámetro con el valor `true`, el comportamiento será el contrario, lo que se conoce como *captura*. Y el primer escuchador que se ejecutará es el del `<body>` y el último el del ``

Probando a añadir ese parámetro a los escuchadores del ejemplo anterior con los divs de colores:

```
divVerde.addEventListener('click', divClick, true );
divRojo.addEventListener('click', divClick, true);
```

```
divAzul.addEventListener('click', divClick, true );
```

Volviendo a hacer clic en el div Azul: veremos un resultado así:

```
Has pulsado: divVerde. Fase: 1, this: divVerde, event.target: divAzul, event.currentTarget: divVerde
Has pulsado: divRojo. Fase: 1, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
```

Observamos que primero responden Verde y Rojo en fase 1 (captura), y por último Azul en fase 2 (objetivo).

En cualquier momento podemos evitar que se siga propagando el evento ejecutando el método `.stopPropagation()` en el código de cualquiera de los escuchadores.

Se pueden ver las distintas fases de un evento en la página domevents.dev.

innerHTML y escuchadores de eventos

Si cambiamos la propiedad `innerHTML` de un elemento del árbol DOM, todos sus escuchadores de eventos desaparecen ya que es como si se volviera a crear ese elemento (y los escuchadores deben ponerse después de crearse).

Por ejemplo: tenemos una tabla de datos y queremos que al hacer doble click en cada fila se muestre su id. La función que añade una nueva fila podría ser:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = `<tr id="${data.id}"><td>${data.dato1}</td><td>${data.dato2}...</td></tr>`;
  miTabla.innerHTML += nuevaFila;
  document.getElementById(data.id).addEventListener('dblclick', event => alert('Id: ' +
    event.target.id));
}
```

Sin embargo, esto sólo funcionaría para la última fila añadida ya que la línea `miTabla.innerHTML += nuevaFila` equivale a `miTabla.innerHTML = miTabla.innerHTML + nuevaFila`. Por tanto, estamos asignando a `miTabla` un código HTML que ya no contiene escuchadores, excepto el de `nuevaFila` que lo ponemos después de hacer la asignación.

La forma correcta de hacerlo sería:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = document.createElement('tr');
  nuevaFila.id = data.id;
  nuevaFila.innerHTML = `<td>${data.dato1}</td><td>${data.dato2}...</td>`;
  nuevaFila.addEventListener('dblclick', event => alert('Id: ' + event.target.id) );
  miTabla.appendChild(nuevaFila);
}
```

De esta forma además mejoramos el rendimiento, ya que el navegador sólo tiene que renderizar el nodo correspondiente a la nuevaFila.

Si lo hacemos como estaba al principio se deben volver a crear y a renderizar todas las filas de la tabla (con todo lo que hay dentro de miTabla).

Eventos personalizados

También podemos, mediante código, lanzar manualmente cualquier evento sobre un elemento con el método `dispatchEvent()`, e incluso crear eventos personalizados. Por ejemplo:

```
const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', (e) => { /* ... */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

Incluso podemos añadir datos al objeto `event` si creamos el evento con `new CustomEvent()`.

Más información en la [página de MDN](#).



DWEC – Javascript Web Cliente.

JavaScript 07 – Objetos nativos	1
Introducción.....	1
Funciones globales.....	1
Objetos nativos del lenguaje.....	2
Objeto Math	3
Objeto Date.....	4

JavaScript 07 – Objetos nativos

Introducción

En este tema vamos a ver:

- Las funciones globales de Javascript, muchas de las cuales ya hemos visto como *Number()* o *String()*.
- Los objetos nativos que incorpora Javascript y que nos facilitarán el trabajo proporcionándonos métodos y propiedades útiles para no tener que “reinventar la rueda” en nuestras aplicaciones.
- Uno de ellos está el objeto **RegExp** que nos permite trabajar con **expresiones regulares** (son iguales que en otros lenguajes) que nos serán de gran ayuda, sobre todo a la hora de validar formularios y que por eso veremos en la siguiente unidad.

Funciones globales

- *parseInt(valor)*: devuelve el valor pasado como parámetro convertido a entero o *NaN* si no es posible la conversión. Este método es mucho más permisivo que *Number* y convierte cualquier cosa que comience por un número (si encuentra un carácter no numérico detiene la conversión y devuelve lo convertido hasta el momento). Ejemplos:

```
console.log( parseInt(3.84) )           // imprime 3 (ignora los decimales)
console.log( parseInt('3.84') )          // imprime 3
console.log( parseInt('28manzanas') )    // imprime 28
console.log( parseInt('manzanas28') )    // imprime NaN
```

- *parseFloat(valor)*: igual, pero devuelve un número decimal. Ejemplos:

```
console.log( parseFloat(3.84) )           // imprime 3.84
console.log( parseFloat('3.84') )          // imprime 3.84
console.log( parseFloat('3,84') )          // imprime 3 (La coma no es un carácter numérico)
console.log( parseFloat('28manzanas') )    // imprime 28
console.log( parseFloat('manzanas28') )    // imprime NaN
```

- `Number(valor)`: convierte el valor a un número. Es como `parseFloat` pero más estricto. Si no puede convertir todo el valor, devuelve `NaN`. Ejemplos:

```
console.log( Number(3.84) )           // imprime 3.84
console.log( Number('3.84') )          // imprime 3.84
console.log( Number('3,84') )          // imprime NaN (La coma no es un carácter numérico)
console.log( Number('28manzanas') )    // imprime NaN
console.log( Number('manzanas28') )    // imprime NaN
```

- `String(valor)`: convierte el valor pasado en una cadena de texto. Si le pasamos un objeto llama al método `toString()` del objeto. Ejemplos:

```
console.log( String(3.84) )           // imprime '3.84'
console.log( String([24, 3, 12]) )      // imprime '24,3,12'
console.log( {nombre: 'Marta', edad: 26} ) // imprime "[object Object]"
```

- `Boolean(valor)`: convierte el valor pasado a un booleano. Sería el resultado de tenerlo como condición en un `if`. Muchas veces en vez de usar esto usamos la doble negación `!!` que da el mismo resultado. Ejemplos:

```
console.log( Boolean('Hola') )         // Equivaldría a !!'Hola'. Imprime true
console.log( Boolean(0) )               // Equivaldría a !!0. Imprime false
```

- `isNaN(valor)`: devuelve `true` si lo pasado NO es un número o no puede convertirse en un número. Ejemplos:

```
console.log( isNaN(3.84) )            // imprime false
console.log( isNaN('3.84') )          // imprime false
console.log( isNaN('3,84') )          // imprime true (La coma no es un carácter numérico)
console.log( isNaN('28manzanas') )     // imprime true
console.log( isNaN('manzanas28') )     // imprime true
```

- `isFinite(valor)`: devuelve `false` si es número pasado es infinito (o demasiado grande)

```
console.log( isFinite(3.84) )          // imprime true
console.log( isFinite(3.84 / 0) )        // imprime false
```

- `encodeURI(string) / decodeURI(string)`: transforma la cadena pasada a una URL codificada válida, transformando los caracteres especiales que contenga, excepto `,` `/` `:` `@` `&` `=` `+` `$` `#`. Debemos usarla siempre que vayamos a pasar una URL. Ejemplo:
 - Decoded: “`http://domain.com?val=1 2 3&val2=r+y%6`”
 - Encoded: “`http://domain.com?val=1%202%203&val2=r+y%256`”
- `encodeURIComponent(string) / decodeURIComponent(string)`: transforma también los caracteres que no transforma la anterior. Debemos usarla para codificar parámetros, pero no una URL entera. Ejemplo:
 - Decoded: “`http://domain.com?val=1 2 3&val2=r+y%6`”
 - Encoded: “`http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256`”

Objetos nativos del lenguaje

En Javascript casi todo son objetos. Ya hemos visto diferentes objetos:

- `window`

- screen
- navigator
- location
- history
- document

Los 5 primeros se corresponden al modelo de objetos del navegador y *document* se corresponde al modelo de objetos del documento. Todos nos permiten interactuar con el navegador para realizar distintas acciones.

También tenemos los tipos de objetos nativos, que no dependen del navegador. Son:

- Number
- String
- Boolean
- Array
- Function
- Object
- Math
- Date
- RegExp

Además de los tipos primitivos de **número**, **cadena**, **booleano**, **undefined** y **null**, Javascript tiene todos los objetos indicados. Como ya hemos visto, se puede crear un número usando su tipo primitivo (`let num = 5`) o su objeto (`let num = new Number(5)`) pero es mucho más eficiente usar los tipos primitivos. Pero aunque lo creemos usando el tipo de dato primitivo se considera un objeto y tenemos acceso a todas sus propiedades y métodos. Ejemplo: `num.toFixed(2)`

Ya hemos visto las principales propiedades y métodos de [Number](#), [String](#), [Boolean](#) y [Array](#) y aquí vamos a ver los de **Math** y **Date** y en el apartado de validar formularios, las de **RegExp**.

Objeto Math

Proporciona constantes y métodos para trabajar con valores numéricicos:

- constantes: `.PI` (número pi), `.E` (número de Euler), `.LN2` (algoritmo natural en base 2), `.LN10` (logaritmo natural en base 10), `.LOG2E` (logaritmo de E en base 2), `.LOG10E` (logaritmo de E en base 10), `.SQRT2` (raíz cuadrada de 2), `.SQRT1_2` (raíz cuadrada de 1/2). Ejemplos:

```
console.log( Math.PI )           // imprime 3.141592653589793
console.log( Math.SQRT2 )         // imprime 1.4142135623730951
```

- `Math.round(x)`: redondea x al entero más cercano
- `Math.floor(x)`: redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- `Math.ceil(x)`: redondea x hacia arriba (5.01 → 6)
- `Math.min(x1,x2,...)`: devuelve el número más bajo de los argumentos que se le pasan.
- `Math.max(x1,x2,...)`: devuelve el número más alto de los argumentos que se le pasan.
- `Math.pow(x, y)`: devuelve x^y (x elevado a y).
- `Math.abs(x)`: devuelve el valor absoluto de x.
- `Math.random()`: devuelve un número decimal aleatorio entre 0 (incluido) y 1 (no incluido).

Si queremos un número entre otros rangos haremos: `Math.random() * (max - min) + min`
O si lo queremos sin decimales, haremos: `Math.round(Math.random() * (max - min) + min)`

- `Math.cos(x)`: devuelve el coseno de x (en radianes).
- `Math.sin(x)`: devuelve el seno de x.
- `Math.tan(x)`: devuelve la tangente de x.
- `Math.sqrt(x)`: devuelve la raíz cuadrada de x

Ejemplos:

```
console.log( Math.round(3.14) )      // imprime 3
console.log( Math.round(3.84) )      // imprime 4
console.log( Math.floor(3.84) )     // imprime 3
console.log( Math.ceil(3.14) )      // imprime 4
console.log( Math.sqrt(2) )         // imprime 1.4142135623730951
```

Objeto Date

Es la clase que usaremos siempre que vayamos a trabajar con fechas. Al crear una instancia de la clase le pasamos la fecha que queremos crear o lo dejamos en blanco para que nos cree la fecha actual. Si le pasamos la fecha podemos pasarle:

- milisegundos, desde la fecha EPOCH
- cadena de fecha
- valor para año, mes (entre 0 y 11), día, hora, minutos, segundos, milisegundos

Ejemplos:

```
let date1=new Date()      // Mon Jul 30 2018 12:44:07 GMT+0200 (CEST) (es cuando he ejecutado la instrucción)
let date7=new Date(1532908000000)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (miliseg. desde 1/1/1070)
let date2=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST) (La fecha pasada a las 0h. GMT)
let date3=new Date('2018-07-30 05:30') // Mon Jul 30 2018 05:30:00 GMT+0200 (CEST) (La fecha pasada a las 05:30h. Local)
let date6=new Date('07-30-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (OJO: formato MM-DD-AAAA)
let date7=new Date('30-Jul-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (tb. podemos poner 'Julio')
let date4=new Date(2018,7,30)       // Thu Ago 30 2018 00:00:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb... y a las 0h. Local)
let date5=new Date(2018,7,30,5,30)  // Thu Ago 30 2018 05:30:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb, ...)
```

EJERCICIO: Crea en la consola dos variables fecNac1 y fecNac2 que contengan tu fecha de nacimiento. La primera la creas pasando una cadena y la segunda pasando año, mes y día.

Cuando ponemos la fecha como texto, como separador de las fechas podemos usar `,` o `espacio`.

Como separador de las horas debemos usar `:`

Cuando ponemos la fecha como parámetros numéricos (separados por `,`) podemos poner valores fuera de rango que se sumarán al valor anterior. Por ejemplo:

```
let date=new Date(2018,7,41)    // Mon Sep 10 2018 00:00:00 GMT+0200 (CEST) -> 41=31Ago+10
let date=new Date(2018,7,0)     // Tue Jul 31 2018 00:00:00 GMT+0200 (CEST) -> 0=0Ago=31Jul (el anterior)
let date=new Date(2018,7,-1)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) -> -1=0Ago-1=31Jul-1=30Jul
```

OJO con el rango de los meses que empieza en 0->Ene, 1->Feb,...,11->Dic

Tenemos métodos **getter** y **setter** para obtener o cambiar los valores de una fecha:

- **fullYear**: permite ver (*get*) y cambiar (*set*) el año de la fecha:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getFullYear() )    // imprime 2018
fecha.setFullYear(2019)             // Tue Jul 30 2019 02:00:00 GMT+0200 (CEST)
```

- **month**: devuelve/cambia el número de mes, pero recuerda que 0->Ene,...,11->Dic

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getMonth() )        // imprime 6
fecha.setMonth(8)                   // Mon Sep 30 2019 02:00:00 GMT+0200 (CEST)
```

- **date**: devuelve/cambia el número de día:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )        // imprime 30
fecha.setDate(-2)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
```

- **day**: devuelve el número de día de la semana (0->Dom, 1->Lun, ..., 6->Sáb). Este método NO tiene *setter*:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDay() )         // imprime 1
```

- **hours, minutes, seconds, milliseconds**, : devuelve/cambia el número de hora, minuto, segundo o milisegundo, respectivamente.
- **time**: devuelve/cambia el número de milisegundos desde Epoch (1/1/1970 00:00:00 GMT):

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getTime() )        // imprime 1532908800000
fecha.setTime(1000*60*60*24*25)     // Fri Jan 02 1970 01:00:00 GMT+0100 (CET) (Le hemos
añadido 25 días a Epoch)
```

EJERCICIO: Realiza en la consola los siguientes ejercicios (usa las variables que creaste antes)

- muestra el día de la semana en que naciste
- modifica fecNac1 para que contenga la fecha de tu cumpleaños de este año (cambia sólo el año)
- muestra el día de la semana de tu cumpleaños de este año
- calcula el nº de días que han pasado desde que naciste hasta hoy

Para mostrar la fecha tenemos varios métodos diferentes:

- **.toString()**: “Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)”
- **.toUTCString()**: “Mon, 30 Jul 2018 00:00:00 GMT”
- **.toDateString()**: “Mon, 30 Jul 2018”
- **.toTimeString()**: “02:00:00 GMT+0200 (hora de verano de Europa central)”
- **.toISOString()**: “2018-07-30T00:00:00.000Z”
- **.toLocaleString()**: “30/7/2018 2:00:00”
- **.toLocaleDateString()**: “30/7/2018”
- **.toLocaleTimeString()**: “2:00:00”

EJERCICIO: muestra en distintos formatos la fecha y la hora de hoy

NOTA: recuerda que las fechas son objetos y que se copian y se pasan como parámetro por referencia:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=fecha
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 28 porque fecha y otraFecha son el mismo
objeto
```

Una forma sencilla de copiar una fecha es crear una nueva pasándole la que queremos copiar:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=new Date(fecha)
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 30
```

En realidad lo que le estamos pasando es el tiempo Epoch de la fecha (es como hacer otraFecha=new Date(fecha.getTime()))

NOTA: la comparación entre fechas funciona correctamente con los operadores >, >=, < y <=

Pero NO funciona con ==, ===, != y !== ya que compara los objetos y ve que son objetos diferentes.

Si queremos saber si 2 fechas son iguales (siendo diferentes objetos):

- el código que pondremos NO es fecha1 === fecha2
- pondremos fecha1.getTime() === fecha2.getTime().

EJERCICIO: comprueba si es mayor tu fecha de nacimiento o el 1 de enero de este año.

Podemos probar los distintos métodos de las fechas en la página de [w3schools](#).